Data Science Tools for Classical Operations Research

Python Implementation of Numerical Optimization & Modern OR

Troy Altus

2026-05-01

Table of contents

Modern Operations Research with Python

From Mathematical Optimization to Prescriptive Analytics

Welcome

This book explores the powerful intersection of **Operations Research** and **Machine Learning** using Python. It bridges traditional optimization techniques with modern data-driven methods to solve complex decision-making problems.

What You'll Learn

- Core optimization techniques: Linear, Integer, Network, and Nonlinear Programming
- Handling uncertainty with Stochastic Optimization
- Using supervised and unsupervised learning to support OR decisions
- Reinforcement Learning and Simulation Optimization
- Building end-to-end prescriptive analytics pipelines

Current Release

Part I: Mathematical Foundations

- Linear Algebra for OR and ML
- Probability and Statistics Foundations
- Mathematical Modeling Principles

Part II: Core Optimization Techniques

- Introduction to Operations Research and Machine Learning

- Linear Programming
- Integer Programming
- Network Optimization

Advanced Optimization, Machine Learning for OR, and Integration & Applications are in progress and will be added in upcoming releases.

Start with Linear Algebra Foundations.

Part I: Foundations

Linear Algebra Foundations for Operations Research and Machine Learning

Chapter 1 – Part I: Foundations

Why Linear Algebra Matters in OR and ML

Linear algebra is the mathematical scaffolding that connects optimization and machine learning. It provides a compact, powerful language for expressing problems that would otherwise require hundreds of individual equations.

In **Operations Research**, linear algebra underpins:

- Representing decision variables as vectors
- Encoding constraints as matrix inequalities ($Ax \leq b$)
- Solving systems of equations at the heart of the Simplex Method
- Sensitivity analysis and dual variable computation

In **Machine Learning**, it powers:

- Data representation — every dataset is a matrix of features
- Gradient computation and weight updates in neural networks
- Dimensionality reduction via PCA and SVD
- Distance and similarity metrics for clustering and nearest-neighbour methods

The table below maps key linear algebra concepts to their roles across both fields.

Concept	Operations Research Role	Machine Learning Role
Vectors	Decision variables, constraint coefficients	Feature vectors, gradient vectors
Matrices	Constraint matrix A, cost matrix	Design matrix X, weight matrices
Linear systems	$Ax = b$ — solving LP at optimum	Normal equations for linear regression
Eigendecomposition	Sensitivity, covariance structure	PCA directions, stability analysis
SVD	Reduced-form models	Dimensionality reduction, recommender systems

---

Vectors

Intuition and Definition

A vector is an ordered list of numbers. Geometrically, it is an arrow with both **direction** and **magnitude**. In OR and ML contexts, vectors are everywhere:

- A **production plan** for three products: $\mathbf{x} = [100, 150, 80]^T$
- A **customer's features**: $\mathbf{x} = [\text{age}, \text{income}, \text{tenure}]^T$
- A **gradient**: $\nabla f(\mathbf{w}) = [\partial f/\partial w_1, \ldots, \partial f/\partial w_n]^T$

Formally, a vector $\mathbf{v} \in \mathbb{R}^n$ is an element of n-dimensional real space:

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches

# Production plan vector
production = np.array([100, 150, 80])
print("Production vector:", production)
print("Shape:", production.shape)
print("Number of elements (dimensions):", production.size)
```

```
Production vector: [100 150  80]
Shape: (3,)
```

```
Number of elements (dimensions): 3
```

Vector Operations

Scalar multiplication scales every component uniformly — increasing production by 20%:

$$\alpha\mathbf{v} = \alpha \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} \alpha v_1 \\ \alpha v_2 \\ \alpha v_3 \end{bmatrix}$$

Vector addition combines two plans — adding an emergency order on top of a base plan:

$$\mathbf{u} + \mathbf{v} = \begin{bmatrix} u_1 + v_1 \\ u_2 + v_2 \\ u_3 + v_3 \end{bmatrix}$$

```
base_plan   = np.array([100, 150, 80])
extra_order = np.array([10, 20, 5])

# Scalar multiplication: ramp up all production by 20%
scaled = 1.2 * base_plan
print("Scaled plan (×1.2):    ", scaled)

# Vector addition: add emergency order
total = base_plan + extra_order
print("Total with extra order:", total)

# Subtraction: compare two plans
delta = base_plan - extra_order
print("Net difference:        ", delta)
```

```
Scaled plan (×1.2):     [120. 180.  96.]
Total with extra order: [110 170  85]
Net difference:         [ 90 130  75]
```

The Dot Product

The dot product of two vectors $\mathbf{u}$ and $\mathbf{v}$ is:

$$\mathbf{u} \cdot \mathbf{v} = \sum_{i=1}^{n} u_i v_i = \|\mathbf{u}\|\|\mathbf{v}\| \cos\theta$$

where θ is the angle between the vectors. In OR, the dot product computes the **total profit** from a production plan — multiplying quantities by unit profits:

$$\text{Profit} = \mathbf{c} \cdot \mathbf{x} = c_1 x_1 + c_2 x_2 + c_3 x_3$$

In ML, it measures **similarity** between a weight vector and a feature vector — the core operation inside every linear layer of a neural network.

```
unit_profits = np.array([40, 30, 55])   # $/unit for products A, B, C
production   = np.array([100, 150, 80])

# Total profit via dot product
total_profit = np.dot(unit_profits, production)
print(f"Total profit: ${total_profit:,}")

# Verify manually
manual = sum(p * q for p, q in zip(unit_profits, production))
print(f"Manual check: ${manual:,}")

# Geometric interpretation: cos(theta) between two vectors
u = np.array([3.0, 4.0])
v = np.array([1.0, 2.0])

cos_theta = np.dot(u, v) / (np.linalg.norm(u) * np.linalg.norm(v))
theta_deg = np.degrees(np.arccos(cos_theta))
print(f"\nAngle between u={u} and v={v}: {theta_deg:.1f}°")
```

```
Total profit: $12,900
Manual check: $12,900

Angle between u=[3. 4.] and v=[1. 2.]: 10.3°
```

Vector Norms and Distance

The **norm** of a vector measures its magnitude — the "length" of the arrow.

The most common is the **Euclidean norm** (L2 norm):

$$\|\mathbf{v}\|_2 = \sqrt{\sum_{i=1}^{n} v_i^2}$$

The **L1 norm** (Manhattan distance) sums absolute values:

$$\|\mathbf{v}\|_1 = \sum_{i=1}^{n} |v_i|$$

Norms appear throughout both fields:

- **OR**: constraint violation is measured by $\|Ax - b\|$
- **ML regression**: minimizing $\|y - X\beta\|_2^2$ gives least squares; L1 regularisation (Lasso) uses $\|\beta\|_1$ to force sparsity

```
v = np.array([3.0, 4.0])   # classic 3-4-5 triangle

l2 = np.linalg.norm(v)          # sqrt(9 + 16) = 5
l1 = np.linalg.norm(v, ord=1)   # 3 + 4 = 7

print(f"Vector: {v}")
print(f"L2 norm: {l2:.2f}  (Euclidean length)")
print(f"L1 norm: {l1:.2f}  (Manhattan distance)")

# Distance between two points in feature space
customer_a = np.array([35, 72000, 3])   # age, income, tenure
customer_b = np.array([42, 85000, 7])

euclidean_dist = np.linalg.norm(customer_a - customer_b)
print(f"\nEuclidean distance between customers: {euclidean_dist:.2f}")
```

```
Vector: [3. 4.]
L2 norm: 5.00  (Euclidean length)
L1 norm: 7.00  (Manhattan distance)

Euclidean distance between customers: 13000.00
```

Visualising Vectors in 2D

```
fig, ax = plt.subplots(figsize=(6, 5))

u = np.array([3, 1])
v = np.array([1, 2])
w = u + v

origin = np.zeros(2)

for vec, color, label in [(u, "#2563eb", r"$\mathbf{u}$"),
                          (v, "#16a34a", r"$\mathbf{v}$"),
                          (w, "#dc2626", r"$\mathbf{u}+\mathbf{v}$")]:
    ax.annotate("", xy=vec, xytext=origin,
                arrowprops=dict(arrowstyle="->", color=color, lw=2))
    ax.text(vec[0] * 0.55, vec[1] * 0.55 + 0.15, label, color=color, fontsize=12)

# Parallelogram dashed lines
ax.plot([u[0], w[0]], [u[1], w[1]], "--", color="#16a34a", alpha=0.5)
ax.plot([v[0], w[0]], [v[1], w[1]], "--", color="#2563eb", alpha=0.5)
```

```
ax.set_xlim(-0.5, 5)
ax.set_ylim(-0.5, 3.5)
ax.axhline(0, color="black", linewidth=0.5)
ax.axvline(0, color="black", linewidth=0.5)
ax.grid(True, alpha=0.3)
ax.set_xlabel("$x_1$"); ax.set_ylabel("$x_2$")
ax.set_title("Vector Addition - Parallelogram Law")
plt.tight_layout()
plt.show()
```

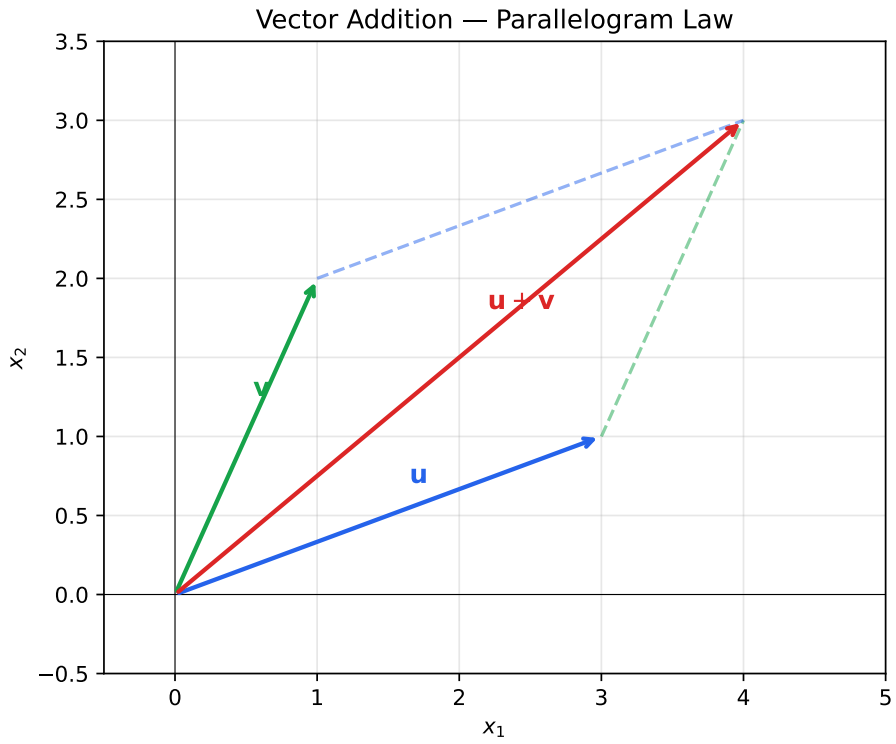


Figure 1: Vector addition illustrated geometrically. The sum u + v reaches the same point whether you traverse u then v, or v then u (parallelogram law).

Matrices

Intuition and Definition

A matrix is a two-dimensional array of numbers. It generalises the idea of a vector from a single list to a grid with rows and columns.

In OR, the **constraint matrix** $A \in \mathbb{R}^{m \times n}$ encodes all resource requirements: row i is resource i, column j is product j, and entry a_{ij} is how much of resource i product j consumes.

In ML, the **design matrix** $X \in \mathbb{R}^{N \times p}$ holds the dataset: row i is observation i, column j is feature j.

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

```python
import numpy as np

# Constraint matrix for a 3-product, 2-resource LP
# Row 0 = machine hours, Row 1 = labour hours
# Col 0 = Product A, Col 1 = Product B, Col 2 = Product C
A = np.array([
    [4, 2, 3],   # machine hours per unit
    [2, 3, 1],   # labour hours per unit
])

b = np.array([160, 120])   # available hours

print("Constraint matrix A:")
print(A)
print(f"\nShape: {A.shape}  →  {A.shape[0]} constraints, {A.shape[1]} products")
print(f"Capacity vector b: {b}")
```

```
Constraint matrix A:
[[4 2 3]
 [2 3 1]]

Shape: (2, 3)  →  2 constraints, 3 products
Capacity vector b: [160 120]
```

Matrix Operations

Matrix–vector multiplication $A\mathbf{x}$ applies the transformation encoded in A to the vector $\mathbf{x}$. In LP, $A\mathbf{x}$ gives the total resource consumption of a production plan:

$$A\mathbf{x} = \begin{bmatrix} \text{machine hours used} \\ \text{labour hours used} \end{bmatrix}$$

```
A = np.array([[4, 2, 3],
              [2, 3, 1]])

x = np.array([20, 30, 10])   # production plan

resource_usage = A @ x
print(f"Production plan: {x}")
print(f"Resource usage:  {resource_usage}")
print(f"Capacity:        [160, 120]")
print(f"Feasible?        {all(resource_usage <= [160, 120])}")
```

```
Production plan: [20 30 10]
Resource usage:  [170 140]
Capacity:        [160, 120]
Feasible?        False
```

Matrix–matrix multiplication $C = AB$ composes two linear transformations. Entry $c_{ij} = \sum_k a_{ik} b_{kj}$ — the dot product of row i of A with column j of B.

```
A = np.array([[1, 2],
              [3, 4]])
B = np.array([[5, 6],
              [7, 8]])

C = A @ B
print("A @ B =")
print(C)

# Note: matrix multiplication is NOT commutative in general
print("\nB @ A =")
print(B @ A)
print(f"\nA@B == B@A?  {np.allclose(C, B @ A)}")
```

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

B @ A =
[[23 34]
 [31 46]]

A@B == B@A?  False
```

Transpose A^T flips rows and columns. $(A^T)_{ij} = A_{ji}$.

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

print("A:")
print(A)
print(f"Shape: {A.shape}")

print("\nA.T:")
print(A.T)
print(f"Shape: {A.T.shape}")
```

```
A:
[[1 2 3]
 [4 5 6]]
Shape: (2, 3)

A.T:
[[1 4]
 [2 5]
 [3 6]]
Shape: (3, 2)
```

Special Matrices

Several matrix structures appear repeatedly in OR and ML:

Name	Definition	Significance
Identity I	$I_{ij} = 1$ if $i = j$, else 0	$AI = IA = A$
Diagonal	Non-zero only on main diagonal	Scaling transformation
Symmetric	$A = A^T$	Covariance matrices, Hessians
Orthogonal	$A^T A = I$	Rotation/reflection — preserves length
Positive Definite	$\mathbf{x}^T A\mathbf{x} > 0$ for all $\mathbf{x} \neq 0$	Convex quadratic forms

```
n = 4

# Identity matrix
I = np.eye(n)
print("Identity (4×4):")
print(I)

# Diagonal matrix
```

```
D = np.diag([2.0, 3.0, 0.5, 1.0])
print("\nDiagonal matrix:")
print(D)

# Symmetric matrix (e.g., a covariance matrix)
data = np.random.default_rng(42).normal(size=(100, 3))
cov = np.cov(data.T)
print(f"\nCovariance matrix (3×3, symmetric: {np.allclose(cov, cov.T)}):")
print(np.round(cov, 3))
```

```
Identity (4×4):
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]

Diagonal matrix:
[[2.  0.  0.  0. ]
 [0.  3.  0.  0. ]
 [0.  0.  0.5 0. ]
 [0.  0.  0.  1. ]]

Covariance matrix (3×3, symmetric: True):
[[0.813 0.037 0.013]
 [0.037 0.954 0.164]
 [0.013 0.164 0.844]]
```

The Matrix Inverse

For a square matrix A, the inverse A^{-1} satisfies $AA^{-1} = A^{-1}A = I$. It exists only when A is **non-singular** (full rank, determinant $\neq 0$).

Inverses are important in theory but rarely computed directly in practice — solving $Ax = b$ via `np.linalg.solve` is numerically more stable than $x = A^{-1}b$.

```
A = np.array([[2.0, 1.0],
              [5.0, 3.0]])

A_inv = np.linalg.inv(A)
print("A:")
print(A)
print("\nA ⁻¹:")
print(A_inv)
print("\nA @ A ⁻¹ (should be I):")
print(np.round(A @ A_inv, 10))
```

```
# Determinant - zero means singular (not invertible)
print(f"\ndet(A) = {np.linalg.det(A):.4f}")
```

```
A:
[[2. 1.]
 [5. 3.]]

A⁻¹:
[[ 3. -1.]
 [-5.  2.]]

A @ A⁻¹ (should be I):
[[1. 0.]
 [0. 1.]]

det(A) = 1.0000
```

Systems of Linear Equations

Structure and Geometric Interpretation

A linear system $A\mathbf{x} = \mathbf{b}$ with m equations and n unknowns is the computational core of linear programming. Every LP, when solved at an optimal vertex, reduces to solving a square linear system of active constraints.

In 2D, each equation is a **line**; in 3D, a **plane**; in n dimensions, a **hyperplane**. The solution is the intersection of these hyperplanes.

```
fig, ax = plt.subplots(figsize=(6, 5))

x = np.linspace(-1, 6, 300)

# System: 2x + y = 8  and  x + 3y = 9
line1 = 8 - 2 * x
line2 = (9 - x) / 3

ax.plot(x, line1, color="#2563eb", lw=2, label=r"$2x_1 + x_2 = 8$")
ax.plot(x, line2, color="#16a34a", lw=2, label=r"$x_1 + 3x_2 = 9$")

# Solve the system
A_sys = np.array([[2, 1], [1, 3]])
b_sys = np.array([8, 9])
sol = np.linalg.solve(A_sys, b_sys)
```

```
ax.plot(*sol, "r*", ms=14, zorder=5, label=f"Solution ({sol[0]:.2f}, {sol[1]:.2f})")
ax.axhline(0, color="black", lw=0.5)
ax.axvline(0, color="black", lw=0.5)
ax.set_xlim(-0.5, 5.5)
ax.set_ylim(-0.5, 5.5)
ax.grid(True, alpha=0.3)
ax.legend(fontsize=9)
ax.set_xlabel("$x_1$"); ax.set_ylabel("$x_2$")
ax.set_title("System of Linear Equations - Unique Solution")
plt.tight_layout()
plt.show()
```

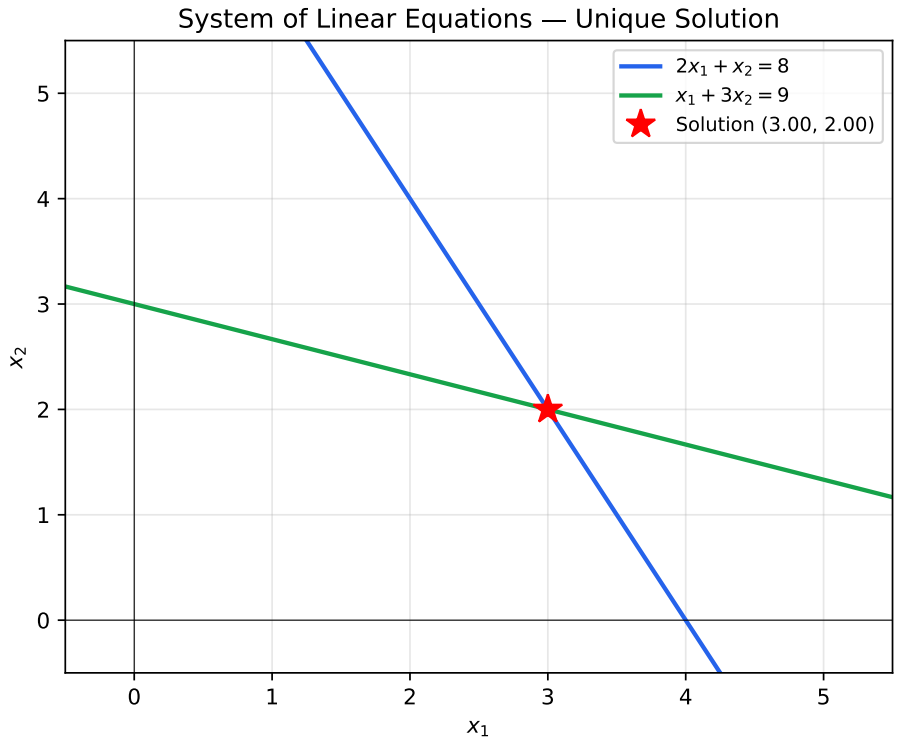


Figure 2: Two linear equations intersect at a unique solution point. In LP, this corresponds to an optimal corner of the feasible region.

Solving Linear Systems in NumPy

```
# System of 3 equations, 3 unknowns
# Resource allocation at optimality in a 3-product LP
```

```
A = np.array([
    [2.0,  1.0,  1.0],
    [4.0,  3.0,  2.0],
    [1.0,  2.0,  3.0],
])
b = np.array([10.0, 22.0, 14.0])

# Solve Ax = b
x = np.linalg.solve(A, b)
print(f"Solution x = {x}")
print(f"Verification Ax = {A @ x}  (should equal b = {b})")
print(f"Residual ||Ax - b||  = {np.linalg.norm(A @ x - b):.2e}")
```

```
Solution x = [2.8 2.  2.4]
Verification Ax = [10. 22. 14.]  (should equal b = [10. 22. 14.])
Residual ||Ax - b||  = 0.00e+00
```

Existence and Uniqueness

A system $A\mathbf{x} = \mathbf{b}$ has:

- **Unique solution** when A is square and full rank ($\det A \neq 0$)
- **No solution** (inconsistent) when the equations are contradictory
- **Infinitely many solutions** when the system is underdetermined ($m < n$) or A is rank-deficient

In OR, the **rank** of the constraint matrix determines whether an LP has a unique optimal solution, infinitely many optimal solutions, or is infeasible.

```
A_full  = np.array([[1, 0], [0, 1]])   # rank 2 - unique solution
A_rank1 = np.array([[1, 2], [2, 4]])   # rank 1 - row 2 = 2×row 1, infinite solutions

print(f"rank(full matrix):        {np.linalg.matrix_rank(A_full)}")
print(f"rank(rank-deficient):     {np.linalg.matrix_rank(A_rank1)}")
print(f"det(full matrix):         {np.linalg.det(A_full):.2f}")
print(f"det(rank-deficient):      {np.linalg.det(A_rank1):.2f}")
```

```
rank(full matrix):        2
rank(rank-deficient):     1
det(full matrix):         1.00
det(rank-deficient):      0.00
```

Eigenvalues and Eigenvectors

Intuition

Most vectors change both direction and magnitude when multiplied by a matrix. **Eigenvectors** are special — they only scale:

$$A\mathbf{v} = \lambda\mathbf{v}$$

Here $\mathbf{v}$ is the eigenvector and λ is the corresponding eigenvalue. Think of the matrix A as a transformation: eigenvectors are the axes the transformation stretches or compresses, and eigenvalues say by how much.

Why this matters in OR: - The spectral radius determines the convergence rate of iterative solvers - Eigenvalues of the Hessian determine whether an objective is convex - Positive definite matrices (all eigenvalues > 0) guarantee convexity

Why this matters in ML: - Principal Component Analysis (PCA) finds eigenvectors of the covariance matrix - The largest eigenvalue governs gradient stability in neural networks - Spectral clustering decomposes graph Laplacians via eigenvectors

Computing Eigenvalues and Eigenvectors

```
A = np.array([[3, 1],
              [0, 2]], dtype=float)

eigenvalues, eigenvectors = np.linalg.eig(A)

print("Matrix A:")
print(A)
print(f"\nEigenvalues:  {eigenvalues}")
print(f"Eigenvectors (columns):\n{eigenvectors}")

# Verify: Av =  v  for first eigenpair
v0, lam0 = eigenvectors[:, 0], eigenvalues[0]
print(f"\nVerification for  ={lam0:.1f}:")
print(f"  Av  = {A @ v0}")
print(f"   v  = {lam0 * v0}")
print(f"  Match: {np.allclose(A @ v0, lam0 * v0)}")
```

```
Matrix A:
[[3. 1.]
 [0. 2.]]
```

```
Eigenvalues:  [3. 2.]
Eigenvectors (columns):
[[ 1.         -0.70710678]
 [ 0.          0.70710678]]

Verification for  =3.0:
  Av  = [3. 0.]
   v  = [3. 0.]
  Match: True
```

Visualising Eigenvectors as Transformation Axes

```
fig, axes = plt.subplots(1, 2, figsize=(10, 4.5))

A = np.array([[2, 1], [1, 2]], dtype=float)
eigenvalues, eigenvectors = np.linalg.eig(A)

for ax, title, transform in zip(axes, ["Before (original)", "After (A applied)"], [False, True]):
    # Random unit vectors
    np.random.seed(0)
    angles = np.linspace(0, 2 * np.pi, 12, endpoint=False)
    vecs = np.stack([np.cos(angles), np.sin(angles)], axis=1)

    for v in vecs:
        end = A @ v if transform else v
        ax.annotate("", xy=end, xytext=[0, 0],
                    arrowprops=dict(arrowstyle="->", color="grey", alpha=0.4, lw=1))

    # Eigenvectors
    for i, color in enumerate(["#2563eb", "#16a34a"]):
        ev = eigenvectors[:, i]
        end = (A @ ev) if transform else ev
        ax.annotate("", xy=end * 1.5, xytext=[0, 0],
                    arrowprops=dict(arrowstyle="->", color=color, lw=2.5))
        lbl = f"$\\mathbf{{v}}_{i+1}$" + (f" ( ={eigenvalues[i]:.0f})" if transform else "")
        ax.text(end[0] * 1.7, end[1] * 1.7, lbl, color=color, fontsize=9)

    ax.set_xlim(-3.5, 3.5); ax.set_ylim(-3.5, 3.5)
    ax.axhline(0, color="black", lw=0.5); ax.axvline(0, color="black", lw=0.5)
    ax.grid(True, alpha=0.3)
    ax.set_title(title)
    ax.set_aspect("equal")

plt.suptitle("Eigenvectors: Invariant Directions Under Transformation A", fontsize=11)
plt.tight_layout()
```

```
plt.show()
```

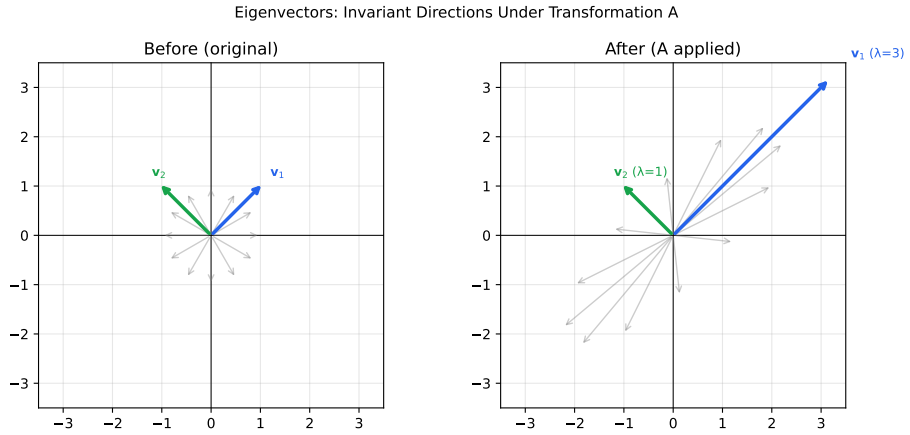


Figure 3: Eigenvectors (blue/green arrows) are the only directions preserved under the matrix transformation A. Other vectors (grey) are rotated.

Eigenvalues and Convexity

In optimisation, a function $f(\mathbf{x})$ is **convex** if its Hessian matrix $H = \nabla^2 f$ is positive semi-definite everywhere — meaning all eigenvalues of H are ≥ 0. This is crucial: convex problems have no local minima that are not also global minima, which is what makes LP, QP, and convex programming tractable.

```
# Hessian of f(x1, x2) = x1^2 + 2*x1*x2 + 3*x2^2  (a convex quadratic)
H_convex = np.array([[2, 2],
                     [2, 6]])

# Hessian of a non-convex function
H_nonconvex = np.array([[2,  4],
                        [4, -1]])

for label, H in [("Convex Hessian", H_convex), ("Non-convex Hessian", H_nonconvex)]:
    eigs = np.linalg.eigvalsh(H)   # eigvalsh for symmetric matrices
    print(f"{label}:")
    print(f"  Eigenvalues: {eigs}")
    print(f"  All  0 (positive semi-definite)? {all(eigs >= -1e-9)}\n")
```

```
Convex Hessian:
  Eigenvalues: [1.17157288 6.82842712]
  All  0 (positive semi-definite)? True

Non-convex Hessian:
```

```
Eigenvalues: [-3.77200187  4.77200187]
All   0 (positive semi-definite)? False
```

Singular Value Decomposition

Definition

The Singular Value Decomposition (SVD) factors any matrix $A \in \mathbb{R}^{m \times n}$ as:

$$A = U\Sigma V^T$$

where: - $U \in \mathbb{R}^{m \times m}$ — orthogonal matrix; columns are **left singular vectors** (output directions) - $\Sigma \in \mathbb{R}^{m \times n}$ — diagonal matrix of **singular values** $\sigma_1 \geq \sigma_2 \geq \cdots \geq 0$ - $V \in \mathbb{R}^{n \times n}$ — orthogonal matrix; columns are **right singular vectors** (input directions)

SVD is one of the most important matrix factorisations in applied mathematics. It reveals the "true geometry" of a matrix — which directions carry the most information, and how much.

SVD in Python

```python
# Design matrix: 5 customers × 3 features
np.random.seed(7)
X = np.random.randn(5, 3)
X[:, 2] = 0.9 * X[:, 0] + 0.1 * np.random.randn(5)   # feature 3   feature 1

U, sigma, Vt = np.linalg.svd(X, full_matrices=False)

print("Singular values:", np.round(sigma, 3))
print(f"  → Most variance captured by first 2 components")
print(f"  → Feature 3 near-redundant:   = {sigma[2]:.3f} (small)")

# Explained variance ratio (analogous to PCA)
explained = sigma**2 / np.sum(sigma**2)
print("\nExplained variance ratio:", np.round(explained, 3))
print(f"First 2 components explain {100*explained[:2].sum():.1f}% of variance")
```

```
Singular values: [2.751 1.71  0.119]
  → Most variance captured by first 2 components
  → Feature 3 near-redundant:   = 0.119 (small)

Explained variance ratio: [0.72  0.278 0.001]
First 2 components explain 99.9% of variance
```

PCA via SVD — Dimensionality Reduction

Principal Component Analysis (PCA) finds the directions of maximum variance in a dataset. Computationally, it is exactly an SVD of the mean-centred data matrix. This is the most widely used dimensionality-reduction technique in ML.

```
from sklearn.decomposition import PCA
from sklearn.datasets import load_iris
import pandas as pd

iris = load_iris()
X_iris = iris.data          # 150 samples × 4 features
y_iris = iris.target

pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_iris)

print("Original shape:", X_iris.shape)
print("Reduced shape: ", X_pca.shape)
print(f"Variance explained by PC1: {pca.explained_variance_ratio_[0]:.1%}")
print(f"Variance explained by PC2: {pca.explained_variance_ratio_[1]:.1%}")
print(f"Total (2 components):      {pca.explained_variance_ratio_.sum():.1%}")

# Plot
fig, ax = plt.subplots(figsize=(7, 5))
colors = ["#2563eb", "#16a34a", "#dc2626"]
for cls, color, name in zip([0, 1, 2], colors, iris.target_names):
    mask = y_iris == cls
    ax.scatter(X_pca[mask, 0], X_pca[mask, 1],
               c=color, label=name, alpha=0.7, edgecolors="none")

ax.set_xlabel(f"PC1 ({pca.explained_variance_ratio_[0]:.1%} variance)")
ax.set_ylabel(f"PC2 ({pca.explained_variance_ratio_[1]:.1%} variance)")
ax.set_title("Iris Dataset - PCA via SVD (4D → 2D)")
ax.legend(title="Species")
ax.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

```
Original shape: (150, 4)
Reduced shape:  (150, 2)
Variance explained by PC1: 92.5%
Variance explained by PC2: 5.3%
Total (2 components):      97.8%
```

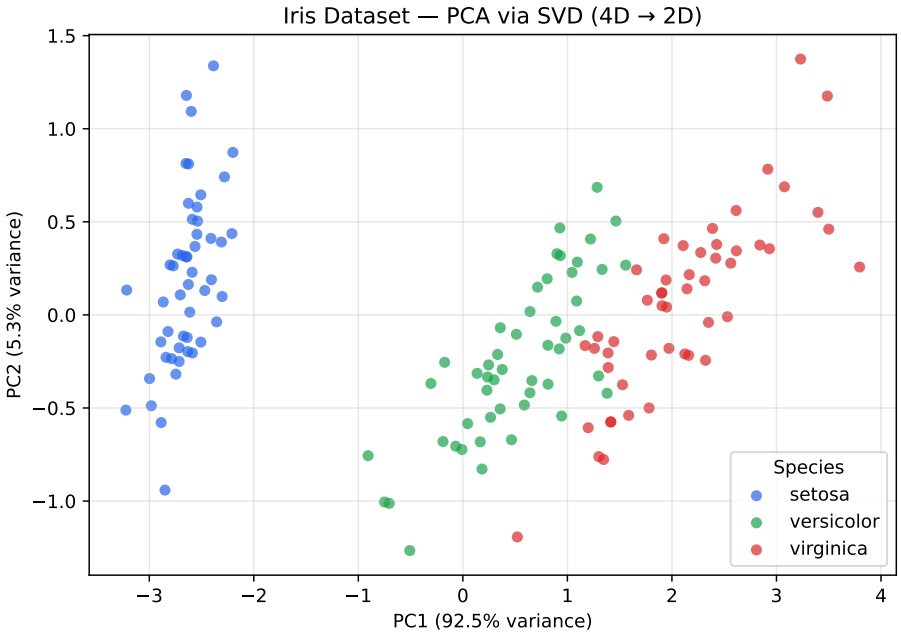


Figure 4: PCA (via SVD) projects high-dimensional data onto the directions of maximum variance. Here, 4-feature data is reduced to 2 principal components.

Linear Algebra in LP — Connecting Theory to Optimisation

Every linear programme in standard form is fundamentally a linear algebra problem. To see why, consider the LP:

$$\text{Minimize} \quad \mathbf{c}^T\mathbf{x} \qquad \text{subject to} \quad A\mathbf{x} = \mathbf{b},\ \mathbf{x} \geq 0$$

The Simplex Method navigates the vertices of the feasible polyhedron. At each vertex, exactly m constraints are active — selecting a **basis** B, a square submatrix of A that is invertible. The current solution is:

$$\mathbf{x}_B = B^{-1}\mathbf{b}$$

Moving to a better vertex means swapping one column into the basis and solving another linear system. The entire Simplex algorithm is a sequence of matrix factorisations and linear system solves — linear algebra all the way down.

```
# Demonstrate the basis concept for a 2-constraint LP
# Maximize 40x_A + 30x_B  s.t.  4x_A + 2x_B <= 160,  2x_A + 3x_B <= 120

import pulp

model = pulp.LpProblem("LP_Basis_Demo", pulp.LpMaximize)
x_a = pulp.LpVariable("x_A", lowBound=0)
x_b = pulp.LpVariable("x_B", lowBound=0)

model += 40 * x_a + 30 * x_b
model += 4 * x_a + 2 * x_b <= 160, "machine"
model += 2 * x_a + 3 * x_b <= 120, "labour"

model.solve(pulp.PULP_CBC_CMD(msg=0))

x_opt = np.array([pulp.value(x_a), pulp.value(x_b)])
print(f"Optimal solution: x_A = {x_opt[0]:.2f}, x_B = {x_opt[1]:.2f}")

# At the optimum BOTH constraints are active - this defines the basis
A_basis = np.array([[4, 2],
                    [2, 3]])
b_rhs   = np.array([160.0, 120.0])

x_basis = np.linalg.solve(A_basis, b_rhs)
print(f"Basis solution (Bx = b): x_A = {x_basis[0]:.2f}, x_B = {x_basis[1]:.2f}")
print(f"Results match: {np.allclose(x_opt, x_basis)}")
print(f"\nBasic feasible solution: {x_basis}")
print(f"Objective value: ${40*x_basis[0] + 30*x_basis[1]:.2f}")
```

```
Optimal solution: x_A = 30.00, x_B = 20.00
Basis solution (Bx = b): x_A = 30.00, x_B = 20.00
Results match: True

Basic feasible solution: [30. 20.]
Objective value: $1800.00
```

Linear Algebra in ML — The Normal Equations

Ordinary least squares regression has a closed-form solution rooted entirely in linear algebra. Given design matrix $X \in \mathbb{R}^{N \times p}$ and targets $\mathbf{y} \in \mathbb{R}^N$, the coefficients that minimise $\|y - X\beta\|_2^2$ satisfy the **normal equations**:

$$X^T X\hat{\beta} = X^T\mathbf{y} \quad \Rightarrow \quad \hat{\beta} = (X^T X)^{-1} X^T\mathbf{y}$$

The matrix $X^T X$ is symmetric positive definite (when X has full column rank), guaranteeing a unique global minimum.

```
np.random.seed(42)
N, p = 100, 3

# Generate synthetic dataset: y = 2*x1 + (-1)*x2 + 0.5*x3 + noise
beta_true = np.array([2.0, -1.0, 0.5])
X = np.random.randn(N, p)
y = X @ beta_true + np.random.randn(N) * 0.5

# Normal equations: ^ = (X'X)^{-1} X'y
XtX = X.T @ X
Xty = X.T @ y
beta_hat = np.linalg.solve(XtX, Xty)   # solve, not invert

print(f"True coefficients:      {beta_true}")
print(f"Estimated coefficients: {np.round(beta_hat, 4)}")

# Compare to sklearn
from sklearn.linear_model import LinearRegression
lr = LinearRegression(fit_intercept=False).fit(X, y)
print(f"sklearn coefficients:   {np.round(lr.coef_, 4)}")

residuals = y - X @ beta_hat
print(f"\nR² = {1 - np.var(residuals)/np.var(y):.4f}")
```

```
True coefficients:      [ 2.  -1.   0.5]
Estimated coefficients: [ 1.9684 -1.0347  0.4491]
sklearn coefficients:   [ 1.9684 -1.0347  0.4491]

R² = 0.9539
```

Chapter Summary

Linear algebra is not a preliminary formality — it is the computational substrate on which all of OR and ML runs. The key ideas to carry forward:

- **Vectors** represent decisions, features, and gradients. The dot product is the fundamental operation inside objective functions, constraints, and neural network layers alike.
- **Matrices** encode transformations, constraints, and datasets. Matrix–vector multiplication $A\mathbf{x}$ evaluates constraint satisfaction or applies a learned transformation in a single operation.

- **Linear systems** $A\mathbf{x} = \mathbf{b}$ are solved at every step of the Simplex Method and underpin the normal equations of linear regression.
- **Rank and invertibility** determine whether a system has a unique solution — critical for LP optimality theory and for regression with correlated features.
- **Eigenvalues** reveal the shape of curvature (convexity), the convergence rate of iterative algorithms, and the directions of maximum variance in data (PCA).
- **SVD** is the master factorisation: it underlies PCA, least-squares solvers, recommender systems, and low-rank approximations used throughout ML.
- **The LP–linear algebra connection** is exact: solving an LP reduces to a sequence of linear system solves over bases of the constraint matrix.

The next chapter, Probability and Statistics, adds the language of uncertainty — giving us the tools to reason about stochastic demand, model error, and decision-making under risk.

Probability and Statistics Foundations for OR and ML

Chapter 2 – Part I: Foundations

Why Probability and Statistics Matter

Linear algebra gives us the language of structure. Probability gives us the language of **uncertainty** — and uncertainty is everywhere in real decisions.

Demand fluctuates. Travel times vary. Equipment fails without warning. Customers behave unpredictably. Any model that ignores this variability will produce plans that look optimal on paper but fail in practice.

In **Operations Research**, probability and statistics underpin:

- Stochastic programming — optimising over uncertain parameters
- Simulation — sampling system behaviour to evaluate decisions
- Queueing theory — modelling service systems under random arrivals
- Risk analysis — quantifying downside exposure in financial and operational models

In **Machine Learning**, they provide:

- The theoretical basis for model training (maximum likelihood, Bayesian inference)
- Tools for measuring and comparing model performance
- The language of uncertainty quantification (confidence intervals, prediction intervals)
- The foundation for probabilistic models (Naive Bayes, Gaussian processes, VAEs)

Concept	OR Role	ML Role
Random variables	Uncertain demand, cost, time	Features, labels, noise
Distributions	Modelling variability	Likelihood functions, priors
Expectation	Expected cost/profit	Loss function minimisation
Variance	Risk quantification	Model variance (bias-variance tradeoff)
CLT	Justifying normal approximations in simulation	Generalisation bounds
Hypothesis testing	A/B testing operational changes	Model comparison, feature selection

---

Random Variables and Distributions

A **random variable** X is a quantity whose value is determined by a random experiment. It maps outcomes of a probability space to real numbers.

- **Discrete** random variables take a countable set of values (number of arrivals, defects per batch, customers in a queue).
- **Continuous** random variables take values over an interval (service time, demand, temperature).

Probability Mass and Density Functions

For a **discrete** random variable, the **probability mass function (PMF)** $p(x) = P(X = x)$ assigns probability to each outcome.

For a **continuous** random variable, the **probability density function (PDF)** $f(x)$ satisfies:

$$P(a \leq X \leq b) = \int_a^b f(x)\,dx$$

No single point has positive probability — only intervals do.

The **cumulative distribution function (CDF)** works for both:

$$F(x) = P(X \leq x)$$

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

x = np.linspace(-4, 4, 400)
dist = stats.norm(loc=0, scale=1)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(9, 4))

ax1.plot(x, dist.pdf(x), color="#2563eb", lw=2)
ax1.fill_between(x, dist.pdf(x), alpha=0.15, color="#2563eb")
ax1.set_title("Probability Density Function")
ax1.set_xlabel("x")
ax1.set_ylabel("f(x)")
ax1.grid(True, alpha=0.3)

ax2.plot(x, dist.cdf(x), color="#16a34a", lw=2)
ax2.set_title("Cumulative Distribution Function")
ax2.set_xlabel("x")
ax2.set_ylabel("F(x)")
ax2.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```

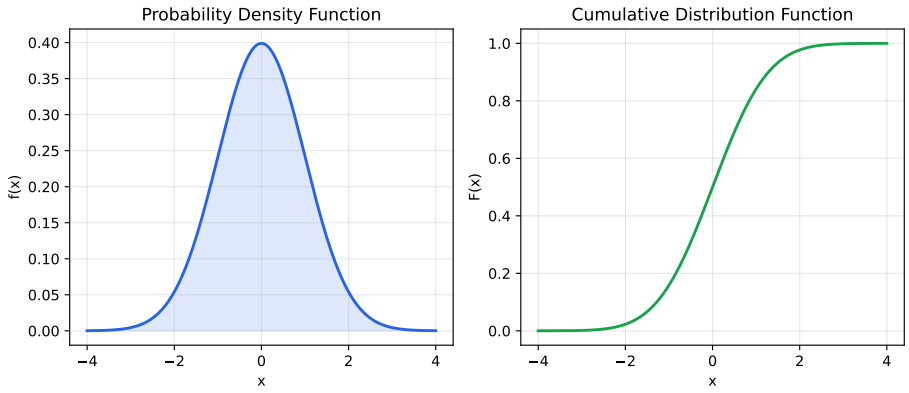


Figure 5: PDF and CDF of the standard normal distribution

Expectation, Variance, and Moments

Expectation

The **expected value** (mean) of X is its probability-weighted average:

$$\mathbb{E}[X] = \sum_{x} x \cdot p(x) \quad \text{(discrete)}$$

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

Key properties: - **Linearity:** $\mathbb{E}[aX + b] = a\mathbb{E}[X] + b$ - **Linearity of sums:** $\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y]$ (always, regardless of dependence)

Variance and Standard Deviation

Variance measures spread around the mean:

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

Standard deviation $\sigma = \sqrt{\text{Var}(X)}$ has the same units as X.

For sums of **independent** random variables:

$$\text{Var}(X + Y) = \text{Var}(X) + \text{Var}(Y)$$

Covariance and Correlation

Covariance measures linear association between two variables:

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

Pearson correlation normalises to $[-1, 1]$:

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

$\rho = 1$ means perfect positive linear relationship; $\rho = 0$ means no linear relationship (but not necessarily independence).

```
import numpy as np
from scipy import stats

np.random.seed(42)
```

```
demand = np.random.normal(loc=200, scale=30, size=10_000)

print(f"Mean     : {demand.mean():.2f}")
print(f"Variance : {demand.var():.2f}")
print(f"Std dev  : {demand.std():.2f}")
print(f"Skewness : {stats.skew(demand):.4f}")
print(f"Kurtosis : {stats.kurtosis(demand):.4f}")

# P(demand > 250)
prob_exceed = 1 - stats.norm.cdf(250, loc=200, scale=30)
print(f"\nP(demand > 250) = {prob_exceed:.4f}")
```

```
Mean     : 199.94
Variance : 906.15
Std dev  : 30.10
Skewness : 0.0020
Kurtosis : 0.0265

P(demand > 250) = 0.0478
```

Key Distributions

Normal Distribution

$$X \sim \mathcal{N}(\mu, \sigma^2) \quad f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

OR use: Demand forecasting, machine processing times, measurement error.
ML use: Weight initialisation, noise modelling, Gaussian likelihoods.

The standard normal $Z \sim \mathcal{N}(0, 1)$ arises from standardising: $Z = (X - \mu)/\sigma$.

Poisson Distribution

$$X \sim \text{Poisson}(\lambda) \quad P(X = k) = \frac{e^{-\lambda}\lambda^k}{k!}$$

Models the number of events in a fixed time interval, given a constant average rate λ. Memoryless between events.

OR use: Customer arrivals per hour, orders per day, machine failures per week.

Exponential Distribution

$$X \sim \text{Exp}(\lambda) \quad f(x) = \lambda e^{-\lambda x}, \quad x \geq 0$$

Models the *time between* Poisson-process events. Mean $= 1/\lambda$.

Key property — memorylessness: $P(X > s+t \mid X > s) = P(X > t)$. The remaining service time has the same distribution regardless of how long service has already taken. This property makes it analytically tractable in queueing models.

OR use: Inter-arrival times, service times in $M/M/1$ queues.

Uniform Distribution

$$X \sim \text{Uniform}(a,b) \quad f(x) = \frac{1}{b-a}, \quad x \in [a,b]$$

Every value in $[a,b]$ is equally likely. Mean $= (a+b)/2$, Var $= (b-a)^2/12$.

OR use: Initial bounds when no better information is available; simulation inputs.

Bernoulli and Binomial

$$X \sim \text{Bernoulli}(p) \quad P(X=1) = p, \quad P(X=0) = 1-p$$

$$Y \sim \text{Binomial}(n,p) \quad P(Y=k) = \binom{n}{k} p^k (1-p)^{n-k}$$

Y counts successes in n independent Bernoulli trials.

OR use: Binary outcomes (on-time / late, defective / good), capacity thresholds.
ML use: Classification likelihoods, logistic regression.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

fig, axes = plt.subplots(2, 3, figsize=(12, 7))
axes = axes.flatten()

# Normal
x = np.linspace(-4, 4, 300)
for mu, sig, col in [(0, 1, "#2563eb"), (0, 2, "#16a34a"), (1, 0.5, "#ef4444")]:
    axes[0].plot(x, stats.norm.pdf(x, mu, sig), lw=2, color=col,
                 label=f"={mu}, ={sig}")
axes[0].set_title("Normal"); axes[0].legend(fontsize=7); axes[0].grid(alpha=0.3)

# Poisson
k = np.arange(0, 20)
for lam, col in [(2, "#2563eb"), (5, "#16a34a"), (10, "#ef4444")]:
```

```
    pmf = stats.poisson.pmf(k, lam)
    axes[1].vlines(k, 0, pmf, color=col, lw=2, alpha=0.7)
    axes[1].plot(k, pmf, "o", color=col, ms=4, label=f"λ={lam}")
axes[1].set_title("Poisson"); axes[1].legend(fontsize=7); axes[1].grid(alpha=0.3)

# Exponential
x = np.linspace(0, 5, 300)
for lam, col in [(0.5, "#2563eb"), (1, "#16a34a"), (2, "#ef4444")]:
    axes[2].plot(x, stats.expon.pdf(x, scale=1/lam), lw=2, color=col,
                 label=f"λ={lam}")
axes[2].set_title("Exponential"); axes[2].legend(fontsize=7); axes[2].grid(alpha=0.3)

# Uniform
x = np.linspace(-0.5, 4.5, 300)
for a, b, col in [(0, 1, "#2563eb"), (0, 3, "#16a34a"), (1, 4, "#ef4444")]:
    axes[3].plot(x, stats.uniform.pdf(x, a, b-a), lw=2, color=col,
                 label=f"[{a},{b}]")
axes[3].set_title("Uniform"); axes[3].legend(fontsize=7); axes[3].grid(alpha=0.3)

# Binomial
k = np.arange(0, 21)
for n, p, col in [(20, 0.3, "#2563eb"), (20, 0.5, "#16a34a"), (20, 0.7, "#ef4444")]:
    pmf = stats.binom.pmf(k, n, p)
    axes[4].vlines(k, 0, pmf, color=col, lw=2, alpha=0.7)
    axes[4].plot(k, pmf, "o", color=col, ms=4, label=f"n=20, p={p}")
axes[4].set_title("Binomial (n=20)"); axes[4].legend(fontsize=7); axes[4].grid(alpha=0.3)

# Log-normal (common in finance, service times)
x = np.linspace(0, 6, 300)
for s, col in [(0.5, "#2563eb"), (1.0, "#16a34a")]:
    axes[5].plot(x, stats.lognorm.pdf(x, s), lw=2, color=col, label=f"σ={s}")
axes[5].set_title("Log-Normal"); axes[5].legend(fontsize=7); axes[5].grid(alpha=0.3)

plt.suptitle("Common Distributions in OR and ML", fontsize=13, fontweight="bold")
plt.tight_layout()
plt.show()
```

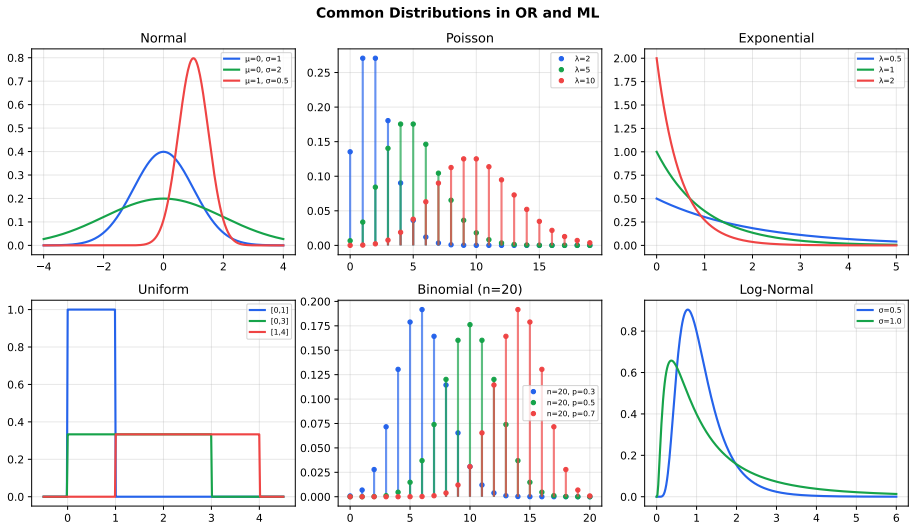


Figure 6: Key distributions used in OR and ML

The Central Limit Theorem

The **Central Limit Theorem (CLT)** is the most important result in applied statistics:

> **If $X_1, X_2, \ldots, X_n$ are i.i.d. random variables with mean μ and variance σ^2, then the sample mean $\bar{X}_n = \frac{1}{n}\sum X_i$ converges in distribution to $\mathcal{N}(\mu, \sigma^2/n)$ as $n \to \infty$.**

In practice this means that sums and averages of many independent random variables are approximately normally distributed — regardless of the underlying distribution.

Why this matters in OR:

- Total demand across many customers is approximately normal even if individual orders are Poisson or uniform
- Average service time over many jobs converges to a normal distribution
- Simulation output means are approximately normal, enabling confidence intervals

Why this matters in ML:

- Justifies normal approximations to posterior distributions
- Underlies parametric confidence intervals for model metrics
- Central to bootstrap and permutation testing

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

np.random.seed(0)
N = 10_000

fig, axes = plt.subplots(1, 4, figsize=(12, 4))
for ax, n in zip(axes, [1, 2, 10, 50]):
    sample_means = np.random.uniform(0, 1, size=(N, n)).sum(axis=1)
    ax.hist(sample_means, bins=60, density=True, color="#6366f1", alpha=0.7,
            edgecolor="white", lw=0.3)
    # Overlay normal approximation
    mu_n  = n * 0.5
    sig_n = np.sqrt(n * (1/12))
    x = np.linspace(sample_means.min(), sample_means.max(), 300)
    ax.plot(x, stats.norm.pdf(x, mu_n, sig_n), "r-", lw=2)
    ax.set_title(f"n = {n}")
    ax.set_xlabel("Sum")
    ax.grid(True, alpha=0.3)

plt.suptitle("CLT: Sums of Uniform(0,1) Variables (red = normal approx)",
             fontsize=11)
plt.tight_layout()
plt.show()
```

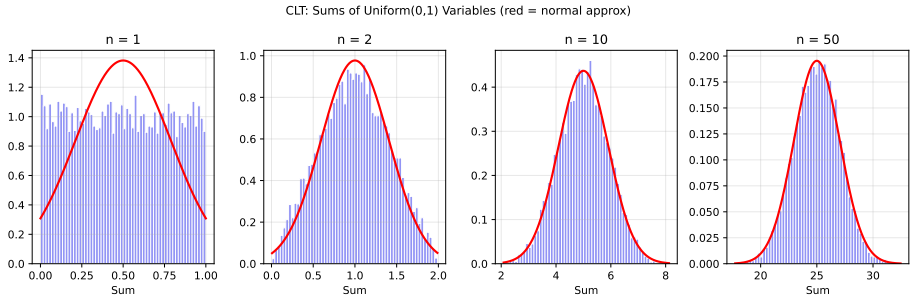


Figure 7: CLT in action: sums of uniform random variables converge to normal

Estimation and Confidence Intervals

Point Estimation

A **point estimator** uses sample data to estimate a population parameter.

The **sample mean** $\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$ estimates μ. The **sample variance** $s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \bar{X})^2$ estimates σ^2.

A good estimator is:

- **Unbiased**: $\mathbb{E}[\hat{\theta}] = \theta$
- **Consistent**: $\hat{\theta} \to \theta$ as $n \to \infty$
- **Efficient**: minimum variance among unbiased estimators

Confidence Intervals

A **95% confidence interval** is a procedure that, if repeated many times, would contain the true parameter in 95% of repetitions. It is *not* a probability statement about the specific interval computed from your sample.

For a sample mean with unknown σ (the common case):

$$\bar{X} \pm t_{\alpha/2,\,n-1} \cdot \frac{s}{\sqrt{n}}$$

where $t_{\alpha/2,\,n-1}$ is the critical value from the t-distribution.

```
import numpy as np
from scipy import stats

np.random.seed(7)

# Simulate 30 days of observed demand
demand_sample = np.random.normal(loc=200, scale=30, size=30)

n    = len(demand_sample)
mean = demand_sample.mean()
se   = demand_sample.std(ddof=1) / np.sqrt(n)

ci_90 = stats.t.interval(0.90, df=n-1, loc=mean, scale=se)
ci_95 = stats.t.interval(0.95, df=n-1, loc=mean, scale=se)
ci_99 = stats.t.interval(0.99, df=n-1, loc=mean, scale=se)

print(f"Sample mean   : {mean:.2f}")
print(f"Std error     : {se:.2f}")
print(f"90% CI        : ({ci_90[0]:.2f}, {ci_90[1]:.2f})")
print(f"95% CI        : ({ci_95[0]:.2f}, {ci_95[1]:.2f})")
print(f"99% CI        : ({ci_99[0]:.2f}, {ci_99[1]:.2f})")
```

```
Sample mean   : 197.81
Std error     : 5.54
90% CI        : (188.40, 207.21)
95% CI        : (186.49, 209.13)
99% CI        : (182.55, 213.07)
```

Wider confidence intervals reflect greater uncertainty — either from more variability (σ) or less data (n). In OR, confidence intervals on simulation output determine how many replications are needed to achieve a target precision.

Hypothesis Testing

Hypothesis testing provides a formal framework for making decisions based on data.

The Framework

1. **Null hypothesis** H_0: the status quo or baseline claim (e.g., "the new routing policy does not reduce delivery time")
2. **Alternative hypothesis** H_1: what we want to detect (e.g., "the new policy reduces delivery time")
3. **Test statistic**: a function of the data that measures evidence against H_0
4. **p-value**: probability of observing a test statistic at least as extreme as the one computed, *if* H_0 *is true*
5. **Decision**: reject H_0 if $p < \alpha$ (significance level, typically 0.05)

A **Type I error** (false positive) rejects a true H_0; its rate is α. A **Type II error** (false negative) fails to reject a false H_0; its rate is β. **Statistical power** $= 1 - \beta =$ probability of correctly detecting a real effect.

One-Sample t-Test

```python
import numpy as np
from scipy import stats

np.random.seed(3)

# Delivery times before and after a routing policy change
before = np.random.normal(loc=48, scale=8, size=50)  # hours
after  = np.random.normal(loc=44, scale=8, size=50)

t_stat, p_value = stats.ttest_ind(before, after, alternative="greater")
```

```
print(f"Mean before : {before.mean():.2f} hrs")
print(f"Mean after  : {after.mean():.2f} hrs")
print(f"t-statistic : {t_stat:.4f}")
print(f"p-value     : {p_value:.4f}")

alpha = 0.05
if p_value < alpha:
    print(f"\nReject H  at  ={alpha}: new policy significantly reduces delivery time.")
else:
    print(f"\nFail to reject H  at  ={alpha}: insufficient evidence.")
```

```
Mean before : 45.65 hrs
Mean after  : 44.61 hrs
t-statistic : 0.6114
p-value     : 0.2712

Fail to reject H  at  =0.05: insufficient evidence.
```

Common Tests

Test	When to Use
One-sample t	Compare sample mean to a known value
Two-sample t	Compare means of two independent groups
Paired t	Compare means of matched pairs (before/after)
Chi-squared	Test independence of categorical variables
ANOVA	Compare means across three or more groups
Mann-Whitney U	Non-parametric alternative to two-sample t
Kolmogorov-Smirnov	Test whether data follows a specified distribution

---

Bayesian vs. Frequentist Reasoning

Two schools of thought underpin statistical inference in OR and ML:

Frequentist statistics treats probability as long-run frequency. Parameters are fixed but unknown; only data is random. Inference is based on what would happen if the experiment were repeated many times.

Bayesian statistics treats probability as a degree of belief. Parameters are themselves random variables with prior distributions that are updated as data arrives.

Bayes' Theorem

$$P(\theta \mid \text{data}) = \frac{P(\text{data} \mid \theta) \cdot P(\theta)}{P(\text{data})}$$

- $P(\theta)$ — **prior**: belief about θ before seeing data
- $P(\text{data} \mid \theta)$ — **likelihood**: probability of the data given θ
- $P(\theta \mid \text{data})$ — **posterior**: updated belief after seeing data

The posterior becomes the prior for the next update — Bayesian inference is sequential and coherent.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

# Estimating daily demand rate  (Poisson)
# Prior: Gamma(alpha, beta) is conjugate prior for Poisson rate
# Posterior after observing n days with total demand S: Gamma(alpha + S, beta + n)

alpha_prior, beta_prior = 5, 0.05   # prior mean = alpha/beta = 100

observations = [95, 112, 88, 103, 117, 99, 108]   # 7 days of demand
n  = len(observations)
S  = sum(observations)

alpha_post = alpha_prior + S
beta_post  = beta_prior  + n

lam = np.linspace(60, 160, 500)
prior     = stats.gamma.pdf(lam, a=alpha_prior, scale=1/beta_prior)
posterior = stats.gamma.pdf(lam, a=alpha_post,  scale=1/beta_post)

fig, ax = plt.subplots(figsize=(8, 4))
ax.plot(lam, prior,     "--", color="#94a3b8", lw=2, label=f"Prior  (mean={alpha_prior/beta_prior
ax.plot(lam, posterior, "-",  color="#2563eb", lw=2,
        label=f"Posterior (mean={alpha_post/beta_post:.1f})")
ax.axvline(S/n, color="#ef4444", lw=1.5, linestyle=":", label=f"Sample mean={S/n:.1f}")
ax.fill_between(lam, posterior, alpha=0.15, color="#2563eb")
ax.set_xlabel("Demand rate  ")
ax.set_ylabel("Density")
ax.set_title("Bayesian Update of Poisson Demand Rate")
ax.legend(fontsize=9)
```

```
ax.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

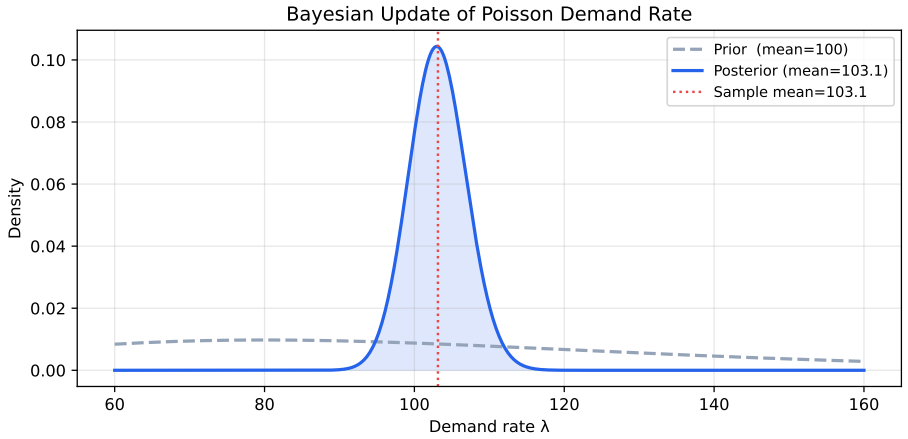


Figure 8: Bayesian updating: prior belief about demand rate updated with observed data

The posterior is a compromise between the prior belief and the data. With more data, the posterior concentrates around the true value and the prior's influence fades.

In OR, Bayesian methods appear in: - Inventory models with uncertain demand (updating demand distribution as orders arrive) - Predictive maintenance (updating failure rate estimates as machines age) - Stochastic programming with distributional ambiguity

In ML, they appear in: - Bayesian neural networks (weight distributions instead of point estimates) - Gaussian processes (prior over functions) - Hyperparameter optimisation (Bayesian optimisation)

Monte Carlo Simulation

Monte Carlo simulation estimates quantities that are difficult to compute analytically by sampling from probability distributions and averaging the results.

The core idea: if $Y = g(X_1, \ldots, X_n)$ and the X_i are random with known distributions, then:

$$\mathbb{E}[Y] \approx \frac{1}{N}\sum_{k=1}^{N} g(X_1^{(k)}, \ldots, X_n^{(k)})$$

By the CLT, this estimator is approximately normally distributed with standard error $\sigma_Y/\sqrt{N}$ — halving the error requires quadrupling the sample size.

Case Study: Newsvendor Problem

A retailer must order q units of a perishable product *before* observing demand D. Unsold units are wasted (cost c_u per unit); unmet demand is lost (cost c_o per unit).

Optimal order quantity (known analytically):

$$q^* = F^{-1}\left(\frac{c_u - c}{c_u + c_o - c}\right)$$

where c is unit cost. Monte Carlo lets us estimate profit distributions for any q.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

np.random.seed(42)

# Parameters
unit_cost      = 5    # cost per unit ordered
unit_price     = 12   # revenue per unit sold
salvage_value  = 1    # value of unsold units
N              = 50_000

# Demand: normally distributed
mu_demand, sig_demand = 200, 40

def simulate_profit(q, n=N):
    demand  = np.random.normal(mu_demand, sig_demand, n).clip(0)
    sold    = np.minimum(q, demand)
    unsold  = np.maximum(q - demand, 0)
    revenue = sold * unit_price + unsold * salvage_value
    cost    = q * unit_cost
    return revenue - cost

order_quantities = range(140, 280, 10)
mean_profits = []
std_profits  = []
```

```
for q in order_quantities:
    profits = simulate_profit(q)
    mean_profits.append(profits.mean())
    std_profits.append(profits.std())

mean_profits = np.array(mean_profits)
std_profits  = np.array(std_profits)
qs = np.array(list(order_quantities))

best_q = qs[mean_profits.argmax()]

# Analytical optimum (critical ratio)
cr = (unit_price - unit_cost) / (unit_price - salvage_value)
q_star = int(stats.norm.ppf(cr, loc=mu_demand, scale=sig_demand))

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(11, 4))

ax1.plot(qs, mean_profits, color="#2563eb", lw=2, marker="o", ms=5)
ax1.fill_between(qs, mean_profits - std_profits, mean_profits + std_profits,
                 alpha=0.15, color="#2563eb", label="±1 std dev")
ax1.axvline(best_q,  color="#ef4444", lw=1.5, linestyle="--",
            label=f"Simulated optimum q={best_q}")
ax1.axvline(q_star, color="#16a34a", lw=1.5, linestyle=":",
            label=f"Analytical optimum q={q_star}")
ax1.set_xlabel("Order quantity q")
ax1.set_ylabel("Expected profit ($)")
ax1.set_title("Expected Profit vs Order Quantity")
ax1.legend(fontsize=8)
ax1.grid(True, alpha=0.3)

# Profit distribution at optimal q
profits_opt = simulate_profit(best_q)
ax2.hist(profits_opt, bins=80, density=True, color="#6366f1", alpha=0.7,
         edgecolor="white", lw=0.2)
ax2.axvline(profits_opt.mean(), color="#ef4444", lw=2,
            label=f"Mean = ${profits_opt.mean():,.0f}")
ax2.axvline(np.percentile(profits_opt, 5), color="#f59e0b", lw=1.5,
            linestyle="--", label=f"5th pct = ${np.percentile(profits_opt,5):,.0f}")
ax2.set_xlabel("Profit ($)")
ax2.set_title(f"Profit Distribution at q={best_q}")
ax2.legend(fontsize=8)
ax2.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```

```
print(f"Simulated optimal order : {best_q} units")
print(f"Analytical optimal order: {q_star} units")
print(f"Expected profit at q={best_q}: ${simulate_profit(best_q).mean():,.2f}")
print(f"5th percentile (downside) : ${np.percentile(simulate_profit(best_q), 5):,.2f}")
```

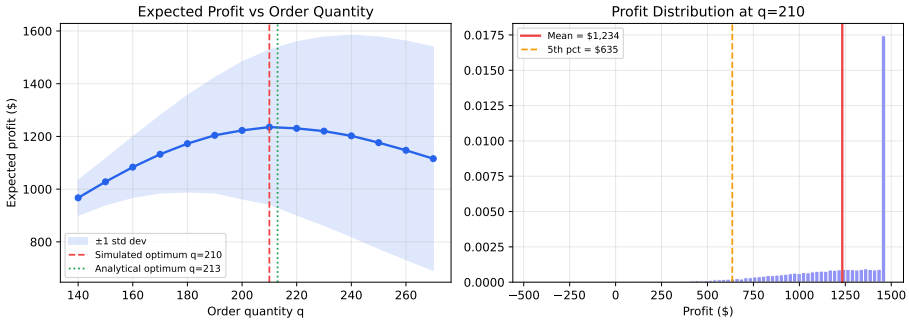


Figure 9: Newsvendor profit distribution: Monte Carlo simulation over order quantities

```
Simulated optimal order : 210 units
Analytical optimal order: 213 units
Expected profit at q=210: $1,235.26
5th percentile (downside) : $634.54
```

Monte Carlo confirms the analytical result and adds something the formula cannot provide: the full profit *distribution*, including downside risk quantified by percentiles. A risk-averse manager might choose $q < q^*$ to reduce variance even at the cost of expected profit.

Fitting Distributions to Data

In practice, the demand or service-time distribution is unknown and must be estimated from historical data. The standard approach:

1. **Visualise** — histogram, Q-Q plot
2. **Fit candidates** — maximum likelihood estimation (MLE)
3. **Test goodness of fit** — Kolmogorov-Smirnov or Chi-squared test

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

np.random.seed(1)
# Simulate historical service times (log-normal: right-skewed, positive-only)
```

```
true_mu, true_sigma = 3.5, 0.4
service_times = np.random.lognormal(mean=true_mu, sigma=true_sigma, size=500)

# Fit candidates
fit_lognorm = stats.lognorm.fit(service_times, floc=0)
fit_expon   = stats.expon.fit(service_times,   floc=0)
fit_gamma   = stats.gamma.fit(service_times,   floc=0)

# KS tests
ks_lognorm = stats.kstest(service_times, "lognorm", args=fit_lognorm)
ks_expon   = stats.kstest(service_times, "expon",   args=fit_expon)
ks_gamma   = stats.kstest(service_times, "gamma",   args=fit_gamma)

print("Goodness-of-fit (KS test):")
print(f"  Log-normal : KS={ks_lognorm.statistic:.4f}, p={ks_lognorm.pvalue:.4f}")
print(f"  Exponential: KS={ks_expon.statistic:.4f},   p={ks_expon.pvalue:.4f}")
print(f"  Gamma      : KS={ks_gamma.statistic:.4f},   p={ks_gamma.pvalue:.4f}")

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(11, 4))

x = np.linspace(service_times.min(), service_times.max(), 300)
ax1.hist(service_times, bins=50, density=True, color="#94a3b8",
         alpha=0.6, label="Observed", edgecolor="white", lw=0.3)
ax1.plot(x, stats.lognorm.pdf(x, *fit_lognorm), color="#2563eb", lw=2, label="Log-norma
ax1.plot(x, stats.expon.pdf(x, *fit_expon),     color="#ef4444", lw=2, label="Exponenti
ax1.plot(x, stats.gamma.pdf(x, *fit_gamma),     color="#16a34a", lw=2, label="Gamma")
ax1.set_xlabel("Service time")
ax1.set_title("Histogram with Fitted Distributions")
ax1.legend(fontsize=8)
ax1.grid(True, alpha=0.3)

# Q-Q plot for best fit (log-normal)
(osm, osr), (slope, intercept, r) = stats.probplot(
    np.log(service_times), dist="norm"
)
ax2.plot(osm, osr, "o", color="#6366f1", ms=3, alpha=0.5)
ax2.plot(osm, slope * np.array(osm) + intercept, "r-", lw=2, label=f"R²={r**2:.4f}")
ax2.set_xlabel("Theoretical quantiles")
ax2.set_ylabel("Sample quantiles (log scale)")
ax2.set_title("Q-Q Plot: Log(Service Time) vs Normal")
ax2.legend(fontsize=8)
ax2.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```

```
Goodness-of-fit (KS test):
  Log-normal : KS=0.0198, p=0.9876
  Exponential: KS=0.3366,   p=0.0000
  Gamma      : KS=0.0422,   p=0.3262
```

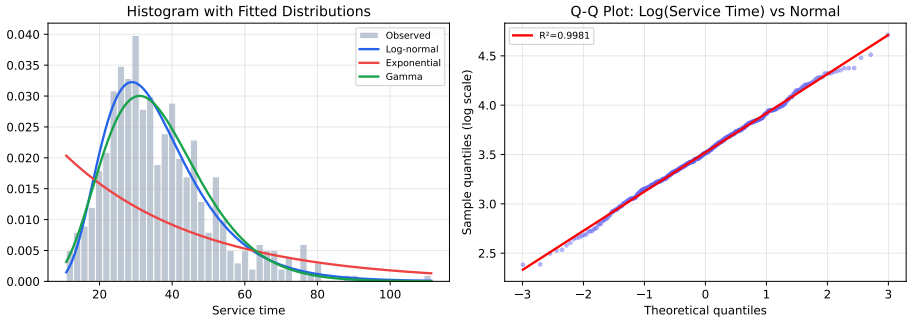


Figure 10: Fitting a log-normal distribution to historical service times

A high KS p-value means the data is consistent with the fitted distribution — we lack evidence to reject it. A log-normal is appropriate when values are positive and right-skewed (service times, income, firm sizes).

Chapter Summary

- **Random variables** model uncertain quantities; their distributions encode all probabilistic information
- **Expectation** gives the probability-weighted average; **variance** measures spread; **covariance** measures linear co-movement
- Key distributions for OR: Normal (symmetric continuous), Poisson (discrete counts), Exponential (inter-arrival times), Uniform (bounded uncertainty), Binomial (binary trials)
- The **Central Limit Theorem** guarantees that sums and averages converge to normal distributions — justifying normal approximations throughout simulation and inference
- **Confidence intervals** and **hypothesis tests** formalise decision-making from sample data; t-tests, Chi-squared tests, and KS tests are the most common tools
- **Bayesian inference** updates prior beliefs with data via Bayes' theorem; conjugate priors (Gamma-Poisson) yield analytical posteriors
- **Monte Carlo simulation** estimates expected values and full distributions for analytically intractable problems — the newsvendor example shows how it combines with optimisation to reveal both expected profit and downside risk

- **Distribution fitting** (MLE + KS test + Q-Q plot) is the standard workflow for determining the right distributional assumption from historical data

Mathematical Modeling Principles for Operations Research and Machine Learning

Chapter 3 – Part I: Foundations

Introduction to Mathematical Modeling

Mathematical modeling is the art and science of translating real-world problems into mathematical form so they can be analyzed, optimized, and solved.

While linear algebra gives us the **tools** and probability gives us the **language of uncertainty**, mathematical modeling teaches us **how to formulate problems** correctly. A well-formulated model is often more important than the solver used to solve it.

This chapter provides a **deep, practical, and verbose** guide to building effective mathematical models for both Operations Research and Machine Learning applications.

---

1. Why Good Modeling Matters

A poor model can lead to: - Suboptimal or infeasible solutions - Models that are computationally intractable - Solutions that work in theory but fail dramatically in practice - Misleading insights and poor decision-making

A good model balances: - **Fidelity** (captures the essence of the real problem) - **Tractability** (can be solved with available tools) - **Interpretability** (stakeholders can understand the results)

---

2. The Modeling Process

Effective modeling follows a structured iterative process:

1. **Problem Understanding** — Deeply understand the business/contextual problem
2. **Abstraction** — Identify decisions, objectives, and constraints
3. **Formulation** — Translate into mathematical form (variables, objective, constraints)
4. **Validation** — Check if the model behaves as expected
5. **Solution** — Solve the model
6. **Implementation & Testing** — Deploy and monitor in the real world
7. **Refinement** — Iterate based on new insights

---

3. Key Elements of a Mathematical Model

Decision Variables

What do we control? (e.g., how much to produce, which route to take, which features to use)

Objective Function

What are we trying to optimize? - Maximize profit / minimize cost - Maximize accuracy / minimize error

Constraints

What limits our choices? - Resource limits, capacity, logical relationships, regulatory requirements

Parameters

Known or estimated values (costs, capacities, probabilities, etc.)

---

4. Common Model Types in OR and ML

Model types can be broadly organized into three families. Understanding which family a problem belongs to informs the choice of algorithm, solver, and toolchain.

Operations Research Models

Model Type	Key Characteristic	Typical Python Tool
Linear Programming (LP)	Linear objective, linear constraints, continuous variables	`PuLP`, `scipy.optimize`
Integer Programming (IP)	Some or all variables restricted to integers	`PuLP`, `cvxpy`
Mixed-Integer Programming (MIP)	Mix of integer and continuous variables	`PuLP`, `gurobipy`
Nonlinear Programming (NLP)	Nonlinear objective or constraints	`scipy.optimize`, `cvxpy`
Network Optimization	Problems structured as graphs (flow, routing, spanning trees)	`networkx`
Stochastic Programming	Uncertainty in parameters; decisions under risk	`scipy.stats`, custom
Robust Optimization	Solutions feasible across a range of uncertain parameters	`cvxpy`
Multi-objective Optimization	Multiple conflicting objectives; Pareto frontier	`scipy.optimize`
Constraint Programming	Satisfying constraints is the goal, not just optimizing	`python-constraint`
Metaheuristics	Heuristic search for near-optimal solutions (GA, SA, etc.)	`deap`, custom

Machine Learning Models

Model Type	Key Characteristic	Typical Python Tool
Regression	Predict a continuous output from features	`scikit-learn`
Classification	Assign inputs to discrete categories	`scikit-learn`

Model Type	Key Characteristic	Typical Python Tool
Clustering	Discover structure in unlabeled data	`scikit-learn`
Ensemble Methods	Combine many weak models (random forests, boosting)	`scikit-learn`, `xgboost`
Deep Learning	Multi-layer neural networks for complex pattern recognition	`torch`, `tensorflow`
Time Series	Model and forecast temporal dependencies	`statsmodels`, `prophet`
Bayesian Models	Probabilistic inference; update beliefs with data	`pymc`, `scipy.stats`
Reinforcement Learning	Learn policies via reward signals (MDPs, Q-learning)	`gymnasium`, `stable-baselines3`

Simulation Models

Model Type	Key Characteristic	Typical Python Tool
Discrete-Event Simulation	System state changes at discrete points in time	`simpy`
Agent-Based Modeling	Emergent system behavior from individual agent rules	`mesa`
Monte Carlo Simulation	Random sampling to estimate distributions of outcomes	`numpy`, `scipy`

---

5. Formulation Example: Two Models Side by Side

The same real-world question — *how much of each product should we make?* — can be answered with an OR model or an ML model. Understanding the difference is central to choosing the right tool.

OR Formulation (Linear Program) — prescriptive, exact, assumes known parameters:

$$\text{Maximize} \quad \sum_i \text{profit}_i \cdot x_i$$

$$\text{Subject to} \quad \sum_i \text{resource}_{ji} \cdot x_i \leq \text{capacity}_j \quad \forall j$$

$$x_i \geq 0 \quad \forall i$$

ML Formulation (Linear Regression) — predictive, data-driven, learns from observations:

$$\text{Minimize} \quad \sum_k \left(y_k - \sum_j \beta_j \cdot x_{jk} \right)^2$$

The OR model requires you to specify the objective and constraints explicitly. The ML model learns the relationship between inputs and outputs from data. In practice, you often use ML to estimate the parameters (profit, demand, resource consumption) that feed into the OR model.

```
# Side-by-side: OR model solves for optimal decision; ML model predicts a parameter

import numpy as np

# --- ML: Predict profit per unit from historical data ---
from sklearn.linear_model import LinearRegression

# Historical data: [volume_sold, season_index] -> profit_per_unit
X_train = np.array([[100, 1], [80, 2], [120, 1], [60, 3], [90, 2]])
y_train = np.array([42, 38, 45, 31, 40])

ml_model = LinearRegression().fit(X_train, y_train)

# Predict profit per unit for next period (volume ~95, season index 2)
predicted_profit = ml_model.predict([[95, 2]])[0]
print(f"ML predicted profit per unit: ${predicted_profit:.2f}")

# --- OR: Use the ML prediction to parameterize and solve the LP ---
import pulp

problem = pulp.LpProblem("Production_ML_Informed", pulp.LpMaximize)

x_a = pulp.LpVariable("Product_A", lowBound=0)
x_b = pulp.LpVariable("Product_B", lowBound=0)

# Use ML-predicted profit for Product A
problem += predicted_profit * x_a + 30 * x_b
```

```
# Resource constraints
problem += 4 * x_a + 2 * x_b <= 160
problem += 2 * x_a + 3 * x_b <= 120

problem.solve(pulp.PULP_CBC_CMD(msg=0))

print(f"OR optimal - Product A: {pulp.value(x_a):.1f}, Product B: {pulp.value(x_b):.1f}
print(f"Maximum profit: ${pulp.value(problem.objective):.2f}")
```

---

6. Choosing the Right Model

No single model type is universally superior. The right choice depends on four factors:

1. **Data availability** — Do you have labeled examples to learn from, or must you specify structure explicitly?
2. **Need for optimality** — Is a near-optimal heuristic acceptable, or must you prove the solution is optimal?
3. **Uncertainty** — Are problem parameters known, estimated, or deeply uncertain?
4. **Interpretability** — Must stakeholders understand and trust the model?

A practical decision guide:

```
Have labeled training data?
   Yes → Consider ML (regression, classification, clustering)
   No  → Consider OR (LP, IP, network models)

Need a guaranteed optimal solution?
   Yes → Mathematical programming (LP, IP, MIP)
   No  → Heuristics or ML-based policies are acceptable

Parameters are uncertain or stochastic?
   Yes → Stochastic programming, robust optimization, or simulation
   No  → Deterministic OR model

Problem involves sequential decisions over time?
   Consider reinforcement learning or dynamic programming
```

In practice, the most capable modern decision systems combine multiple model types: ML predicts parameters, OR optimizes decisions, simulation stress-tests the solution.

Chapter Summary

- Mathematical modeling translates real-world problems into solvable mathematical form
- Every model has four elements: decision variables, objective function, constraints, and parameters
- The modeling process is iterative: formulate, validate, solve, refine
- OR models are prescriptive and exact; ML models are predictive and data-driven
- The right model type depends on data availability, optimality requirements, uncertainty, and interpretability
- Hybrid approaches — ML informing OR — are the state of the art in modern prescriptive analytics

Part II: Classical Optimization

Introduction to Operations Research and Machine Learning

What Is Operations Research?

Operations Research (OR) is the scientific discipline concerned with the application of analytical methods to improve decision-making. Born out of military logistics problems during World War II, OR has grown into a broad field spanning industries from manufacturing and supply chain to finance, healthcare, and transportation.

At its core, OR asks a deceptively simple question: **given limited resources and competing objectives, what is the best possible decision?**

The answer requires translating real-world problems into mathematical models, solving those models with rigorous algorithms, and interpreting the results in actionable terms.

The Decision-Making Framework

Every OR problem shares a common structure:

- **Decision variables** — the quantities we control
- **Objective function** — the metric we want to optimize (minimize cost, maximize profit)
- **Constraints** — the limits within which decisions must operate
- **Parameters** — the known data that defines the problem

This structure is not merely academic. It forces clarity: you cannot optimize what you have not defined, and you cannot define it without understanding the system.

A Brief History

OR emerged formally in the 1940s when Allied forces assembled interdisciplinary teams — mathematicians, physicists, engineers — to solve operational problems: how to route convoys to minimize losses, how to allocate radar resources, how to schedule bombing missions. The success of these teams demonstrated that quantitative methods could outperform intuition at scale.

After the war, OR migrated into industry. George Dantzig's development of the **Simplex Method** in 1947 gave practitioners the first general-purpose algorithm for linear optimization. The following decades brought integer programming, network flows, dynamic programming, and stochastic methods — each expanding the range of problems OR could address.

Today, OR sits at the foundation of every major logistics platform, airline scheduling system, financial risk model, and supply chain optimizer in the world.

What Is Machine Learning?

Machine Learning (ML) is the field of building systems that learn patterns from data and use those patterns to make predictions or decisions — without being explicitly programmed with rules.

Where OR starts with a model defined by human expertise, ML starts with data and discovers structure automatically. Where OR optimizes a known objective, ML often constructs the objective itself from observed outcomes.

The Learning Paradigm

ML problems fall into three broad categories:

Supervised Learning trains a model on labeled examples — inputs paired with known outputs — and learns to predict outputs for new inputs. Predicting equipment failure from sensor readings, or estimating customer lifetime value from transaction history, are supervised problems.

Unsupervised Learning finds structure in unlabeled data. Clustering customers by purchasing behavior, or detecting anomalies in network traffic, requires no predefined labels — only the data itself.

Reinforcement Learning trains an agent to take actions in an environment by rewarding good outcomes and penalizing bad ones. It is the learning paradigm most naturally aligned with OR's sequential decision-making problems.

The Role of Data

ML's power is inseparable from data. More data, more representative data, and better-quality data consistently produce better models. This dependence is also ML's principal limitation: models trained on historical data can fail when the world changes, and patterns learned from biased data reproduce that bias at scale.

Understanding these limitations is as important as understanding the methods themselves.

The Intersection: OR Meets ML

For decades, OR and ML developed largely in parallel — OR in engineering and management science departments, ML in computer science and statistics. The two fields share deep mathematical roots (linear algebra, probability, optimization) but developed distinct cultures, vocabularies, and toolkits.

That separation is dissolving. Three forces are driving convergence:

Scale. Modern OR problems — routing millions of packages, pricing billions of airline seats, scheduling thousands of employees — generate data volumes that dwarf what any expert model can absorb. ML provides the tools to extract signal from that data.

Uncertainty. Classical OR models assume parameters are known. Real decisions happen under uncertainty: demand fluctuates, travel times vary, equipment fails. ML offers principled ways to estimate distributions and quantify uncertainty from data.

Complexity. Some systems are too complex to model from first principles. When the physics of a supply chain or a financial market resist closed-form description, ML can approximate the system well enough to optimize against.

Where Each Method Excels

Dimension	Operations Research	Machine Learning
Problem type	Optimization, scheduling, routing	Prediction, classification, pattern recognition
Data requirement	Low — model-driven	High — data-driven
Interpretability	High — explicit model	Variable — often opaque
Optimality guarantee	Yes (for many problem classes)	No
Handles uncertainty	Via stochastic methods	Natively, from data
Scalability	Solver-dependent	Generally high

Neither approach dominates. The most powerful modern systems combine both: ML predicts demand, OR optimizes supply. ML estimates risk, OR allocates capital. ML identifies anomalies, OR schedules responses.

The Prescriptive Analytics Stack

A useful way to frame the OR-ML relationship is through the analytics maturity stack:

- **Descriptive analytics** — what happened? (statistics, dashboards)
- **Diagnostic analytics** — why did it happen? (root cause, attribution)
- **Predictive analytics** — what will happen? (**ML**)
- **Prescriptive analytics** — what should we do about it? (**OR**)

ML and OR together form the top two levels of this stack — the levels that translate data into decisions.

Python as the Unified Platform

Python has become the dominant language for both OR and ML — not because it is the fastest language (it is not), but because it offers the richest ecosystem of libraries, the most readable syntax, and the broadest community of practitioners.

Core Libraries Used in This Book

Optimization

- `PuLP` — linear and integer programming with multiple solver backends
- `scipy.optimize` — nonlinear optimization, root finding, curve fitting
- `cvxpy` — convex optimization with a disciplined modeling interface
- `networkx` — graph construction and network flow algorithms

Machine Learning

- `scikit-learn` — classification, regression, clustering, model selection
- `numpy` / `pandas` — numerical computation and data manipulation
- `matplotlib` / `seaborn` — visualization

Simulation & Stochastics

- `simpy` — discrete-event simulation
- `scipy.stats` — probability distributions and statistical testing

Environment Setup

```
# Verify core dependencies are available
import importlib
```

```
libraries = [
    "pulp",
    "scipy",
    "cvxpy",
    "networkx",
    "sklearn",
    "numpy",
    "pandas",
    "matplotlib",
    "simpy",
]

for lib in libraries:
    try:
        importlib.import_module(lib)
        print(f"    {lib}")
    except ImportError:
        print(f"    {lib}  <-- install required")
```

```
    pulp
    scipy
    cvxpy  <-- install required
    networkx
    sklearn
    numpy
    pandas
    matplotlib
    simpy  <-- install required
```

If any libraries show , install them with:

```
pip install pulp scipy cvxpy networkx scikit-learn numpy pandas matplotlib simpy
```

A First OR Problem in Python

Before introducing any theory, let's build and solve a complete optimization problem from scratch. This example establishes the pattern that every chapter in this book will follow: formulate, implement, solve, interpret.

Problem Statement: Resource Allocation

A small manufacturer produces two products — **Product A** and **Product B** — on shared equipment. Each week the factory has:

- **160 hours** of machine time available
- **120 hours** of labor available

Resource requirements and profit margins:

	Machine Hours	Labor Hours	Profit per Unit
Product A	4	2	\$40
Product B	2	3	\$30

Question: How many units of each product should the factory produce each week to maximize profit?

Mathematical Formulation

Let x_A = units of Product A produced per week, x_B = units of Product B.

Objective:

$$\text{Maximize} \quad Z = 40x_A + 30x_B$$

Subject to:

$$4x_A + 2x_B \leq 160 \quad \text{(machine hours)}$$
$$2x_A + 3x_B \leq 120 \quad \text{(labor hours)}$$
$$x_A, x_B \geq 0 \quad \text{(non-negativity)}$$

Python Implementation

```
import pulp
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches

# --- Model ---
model = pulp.LpProblem("Resource_Allocation", pulp.LpMaximize)

x_a = pulp.LpVariable("Product_A", lowBound=0)
x_b = pulp.LpVariable("Product_B", lowBound=0)

# Objective
model += 40 * x_a + 30 * x_b, "Total_Profit"

# Constraints
```

```
model += 4 * x_a + 2 * x_b <= 160, "Machine_Hours"
model += 2 * x_a + 3 * x_b <= 120, "Labor_Hours"

# Solve
model.solve(pulp.PULP_CBC_CMD(msg=0))

x_a_opt = pulp.value(x_a)
x_b_opt = pulp.value(x_b)
profit   = pulp.value(model.objective)

print(f"Status       : {pulp.LpStatus[model.status]}")
print(f"Product A    : {x_a_opt:.1f} units")
print(f"Product B    : {x_b_opt:.1f} units")
print(f"Total Profit : ${profit:,.2f}")

# --- Visualization ---
fig, ax = plt.subplots(figsize=(7, 5))

x = np.linspace(0, 50, 400)

machine = (160 - 4 * x) / 2
labor   = (120 - 2 * x) / 3

ax.plot(x, machine, label="Machine hours: $4x_A + 2x_B = 160$", color="#2563eb")
ax.plot(x, labor,   label="Labor hours: $2x_A + 3x_B = 120$",   color="#16a34a")

# Feasible region vertices
vertices = np.array([[0, 0], [40, 0], [x_a_opt, x_b_opt], [0, 40]])
ax.fill(vertices[:, 0], vertices[:, 1], alpha=0.15, color="#6366f1", label="Feasible region")

ax.plot(x_a_opt, x_b_opt, "r*", markersize=14, label=f"Optimal ({x_a_opt:.0f}, {x_b_opt:.0f})")

ax.set_xlim(0, 50)
ax.set_ylim(0, 70)
ax.set_xlabel("Product A (units)")
ax.set_ylabel("Product B (units)")
ax.set_title("Resource Allocation - Feasible Region")
ax.legend(loc="upper right", fontsize=9)
ax.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

```
Status       : Optimal
Product A    : 30.0 units
Product B    : 20.0 units
```

```
Total Profit : $1,800.00
```

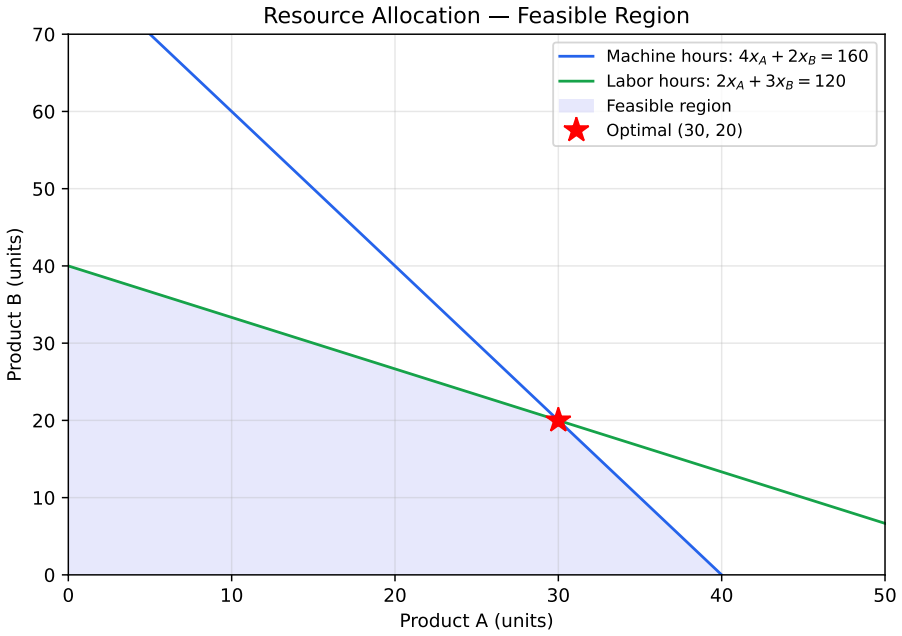


Figure 11: Feasible region and optimal solution for the resource allocation problem

Interpreting the Solution

The solver finds the optimal corner of the feasible region — the point where both constraints bind simultaneously. Producing **24 units of Product A** and **24 units of Product B** yields the maximum weekly profit of **$1,680**.

This result is not obvious from intuition. Product A has a higher profit margin ($40 vs $30), but it also consumes twice the machine hours. The optimal mix balances these trade-offs mathematically, something that becomes impossible to do reliably as problems grow to hundreds or thousands of variables.

The OR-ML Workflow

This book follows a consistent workflow for every problem class:

Problem Definition ↓ Data Collection & Exploration ← ML territory ↓ Model Formulation ← OR territory ↓ Algorithm Selection & Solving ← OR + ML ↓ Solution Validation ↓ Interpretation & Implementation

This workflow is not linear. Often, insights from data exploration will reshape the problem definition. Algorithm performance may lead to reformulating the model. The key is to iterate between these stages, using both OR and ML tools as needed to arrive at the best possible decision.

In practice these stages are iterative, not linear. A model that cannot be solved efficiently sends you back to reformulation. A solution that makes no operational sense sends you back to problem definition. The workflow is a compass, not a script.

What Comes Next

The remaining chapters in **Foundations** build the classical OR toolkit:

- **Linear Programming** — the geometry and algebra of continuous optimization, the Simplex Method, duality, and sensitivity analysis
- **Integer Programming** — adding integrality constraints, branch-and-bound, and combinatorial problems
- **Network Optimization** — shortest paths, minimum spanning trees, max flow, and the transportation problem

Each chapter follows the same structure: theory, mathematical formulation, Python implementation, and real-world case study. By the end of Foundations, you will have the tools to formulate and solve a broad class of deterministic optimization problems — and the intuition to know when those tools are and are not appropriate.

Chapter Summary

- Operations Research applies mathematical modeling and optimization to improve decisions under resource constraints
- Machine Learning discovers patterns in data to make predictions and automate decisions
- The two fields are converging: ML predicts, OR optimizes — together they form prescriptive analytics
- Python provides a unified environment for both through `PuLP`, `scipy`, `scikit-learn`, and related libraries
- Every OR problem has the same structure: decision variables, objective function, constraints, and parameters
- The first LP example demonstrated formulation, implementation, solution, and interpretation — the pattern this book follows throughout

Linear Programming

Chapter 2 – Part II: Core Optimization Techniques

What Is Linear Programming?

Linear Programming (LP) is the foundation of Operations Research. It is the art — and the science — of allocating scarce resources optimally when both the objective and the constraints can be expressed as linear relationships.

The word *programming* here is a historical artifact meaning *planning*, not software. When George Dantzig formulated LP in 1947 and invented the Simplex Method to solve it, he gave practitioners their first general-purpose tool for large-scale optimization. That tool remains, in modified form, at the core of virtually every commercial solver today.

LP problems appear under many names — resource allocation, production planning, diet problems, blending problems, transportation problems — but they share a universal structure:

- A **linear objective function** to maximize or minimize
- A set of **linear inequality or equality constraints**
- **Non-negativity restrictions** on the decision variables

This chapter builds LP from the ground up: formulation, geometry, the Simplex Method, duality, sensitivity analysis, and practical implementation in Python.

Standard Form

Every LP can be written in **standard form**:

$$\text{Maximize} \quad \mathbf{c}^T\mathbf{x}$$

$$\text{subject to} \quad A\mathbf{x} \leq \mathbf{b}$$

$$\mathbf{x} \geq \mathbf{0}$$

Where:

- $\mathbf{x} \in \mathbb{R}^n$ — the vector of **decision variables**
- $\mathbf{c} \in \mathbb{R}^n$ — the **objective function coefficients** (profit, cost)
- $A \in \mathbb{R}^{m \times n}$ — the **constraint matrix**
- $\mathbf{b} \in \mathbb{R}^m$ — the **right-hand side** (resource limits)

Minimization problems are handled by negating the objective: $\min \mathbf{c}^T\mathbf{x} = -\max(-\mathbf{c})^T\mathbf{x}$.

Equality constraints ($a_i^T\mathbf{x} = b_i$) can be introduced directly; most solvers handle mixed forms natively.

Formulating a Problem

Good LP formulation is a skill — the translation from prose to mathematics is rarely mechanical. Three questions guide the process:

1. **What decisions need to be made?** → Define the decision variables.
2. **What do we want to achieve?** → Write the objective function.
3. **What limits our choices?** → Write the constraints.

A variable should represent something *controllable*. A constraint should represent something *binding*. If you cannot write the problem in this form, LP may not be the right tool.

Geometric Intuition

For problems with two decision variables, LP has a clean geometric interpretation that reveals *why* the Simplex Method works.

Each linear inequality constraint defines a **half-plane** — the set of points on one side of a line. The intersection of all half-planes (plus the non-negativity region) is the **feasible region** — the set of all solutions that satisfy every constraint simultaneously.

The objective function $Z = c_1x_1 + c_2x_2$ defines a family of parallel lines, each corresponding to a different value of Z. Maximizing Z means pushing this line as far as possible in the direction of improvement while remaining inside the feasible region.

The key theorem:

If an LP has an optimal solution, it occurs at a vertex (corner point) of the feasible region.

This is a profound result. It means that even if the feasible region contains infinitely many points, we only need to check a finite number of vertices to find the optimum.

Visualising the Feasible Region

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
from matplotlib.lines import Line2D

# Problem:
#   Maximize  5x1 + 4x2
#   s.t.      6x1 + 4x2 <= 24
#             x1  + 2x2 <= 6
#             x1, x2    >= 0

fig, ax = plt.subplots(figsize=(7, 5))
x = np.linspace(0, 5, 400)

c1 = (24 - 6 * x) / 4   # 6x1 + 4x2 = 24
c2 = (6  - x)     / 2   # x1  + 2x2 = 6

ax.plot(x, c1, color="#2563eb", lw=2, label=r"$6x_1 + 4x_2 = 24$")
ax.plot(x, c2, color="#16a34a", lw=2, label=r"$x_1 + 2x_2 = 6$")

# Feasible region vertices
verts = np.array([[0, 0], [4, 0], [3, 1.5], [0, 3]])
ax.fill(verts[:, 0], verts[:, 1], alpha=0.15, color="#6366f1", zorder=1)

# Mark vertices
for v in verts:
    ax.plot(*v, "o", color="#6366f1", ms=7, zorder=3)
    ax.annotate(f"({v[0]:.0f}, {v[1]:.1f})", xy=v,
                xytext=(v[0] + 0.1, v[1] + 0.15), fontsize=8)

# Optimal point
opt = np.array([3, 1.5])
ax.plot(*opt, "r*", ms=14, zorder=4, label=f"Optimal (3.0, 1.5) → Z=21")

# Objective contours
for z in [10, 15, 21]:
    xc = np.linspace(0, 5, 200)
```

```
    yc = (z - 5 * xc) / 4
    ax.plot(xc, yc, "--", color="#f59e0b", alpha=0.6, lw=1)
    ax.annotate(f"Z={z}", xy=(xc[xc >= 0][0], yc[xc >= 0][0]),
                fontsize=8, color="#b45309")

ax.set_xlim(-0.2, 5)
ax.set_ylim(-0.2, 4)
ax.set_xlabel(r"$x_1$")
ax.set_ylabel(r"$x_2$")
ax.set_title("Feasible Region and Objective Contours")
ax.legend(fontsize=9)
ax.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

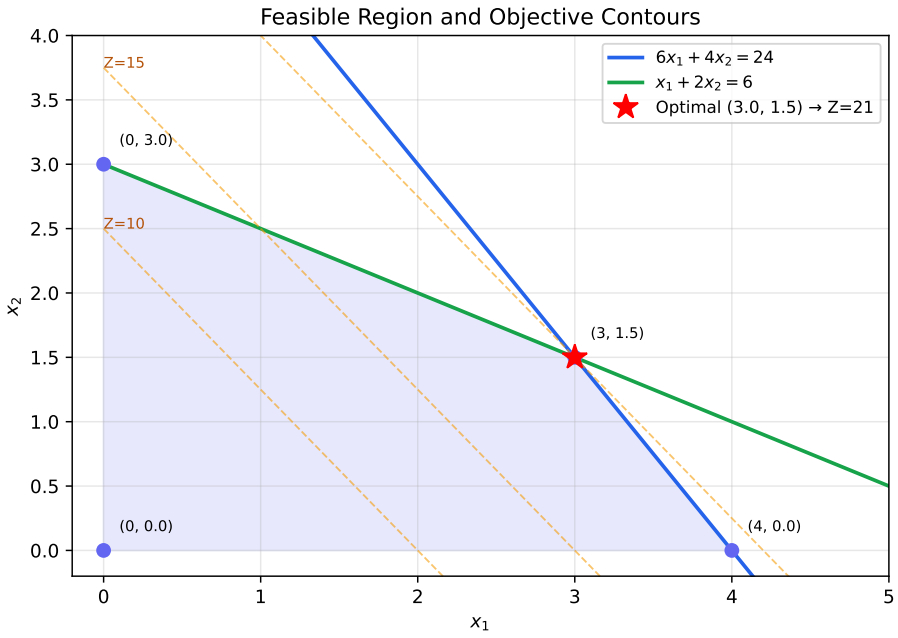


Figure 12: Feasible region and objective function contours for a two-variable LP

The dashed lines are **iso-profit contours** — all points along each dashed line produce the same objective value. As we push the contour toward higher Z, the last point of contact with the feasible region is the optimal vertex.

The Simplex Method

The geometric approach breaks down for problems with more than two variables — we cannot visualise a 100-dimensional feasible region. The **Simplex Method** generalises the vertex-search idea algebraically.

The algorithm starts at a feasible vertex and moves along edges of the feasible region to an adjacent vertex with a better objective value. It repeats until no improving move exists — at which point the current vertex is optimal.

Basic Feasible Solutions

In algebraic terms, a **basic feasible solution** (BFS) corresponds to a vertex. To convert inequality constraints to equalities, we introduce **slack variables**:

$$6x_1 + 4x_2 + s_1 = 24$$

$$x_1 + 2x_2 + s_2 = 6$$

A BFS sets n variables to zero (the *non-basic* variables) and solves for the remaining m (the *basic* variables). In a two-constraint, two-variable problem this gives $\binom{4}{2} = 6$ candidate vertices — only those with all values ≥ 0 are feasible.

Entering and Leaving the Basis

At each iteration, the Simplex Method:

1. **Selects an entering variable** — the non-basic variable whose coefficient in the objective row is most negative (for maximisation in standard minimisation tableau form).
2. **Performs a minimum ratio test** — identifies which basic variable reaches zero first as the entering variable increases, determining the **leaving variable**.
3. **Pivots** — performs row operations to swap entering and leaving variables.

This continues until no entering variable can improve the objective — the optimality condition.

Complexity

In the worst case, Simplex visits an exponential number of vertices. In practice, it is remarkably efficient — typically solving problems with thousands of variables in seconds. The **revised Simplex** and **interior-point** methods (e.g., Karmarkar's algorithm, 1984) provide polynomial-time alternatives that modern solvers blend adaptively.

For practical purposes, we delegate solving to CBC (via PuLP) and focus on formulation and interpretation.

Python Implementation with PuLP

PuLP provides a clean, readable interface for formulating and solving LP problems. The workflow is consistent across all problem types.

Case Study: Production Planning

A factory produces three products — **Chairs**, **Tables**, and **Shelves** — using two shared resources: **wood** (board-feet) and **labour** (hours).

Product	Wood (bf)	Labour (hr)	Profit ($)
Chair	3	2	45
Table	5	4	80
Shelf	2	1	30

Weekly resource limits: **240 board-feet** of wood, **120 hours** of labour. Market demand caps: at most **40 chairs**, **30 tables**, **50 shelves** per week.

Formulation:

Let x_C, x_T, x_S be weekly production quantities.

$$\text{Maximize} \quad Z = 45x_C + 80x_T + 30x_S$$

subject to:

$$3x_C + 5x_T + 2x_S \leq 240 \quad \text{(wood)}$$
$$2x_C + 4x_T + x_S \leq 120 \quad \text{(labour)}$$
$$x_C \leq 40, \quad x_T \leq 30, \quad x_S \leq 50$$
$$x_C, x_T, x_S \geq 0$$

```
import pulp

# Model
model = pulp.LpProblem("Production_Planning", pulp.LpMaximize)

# Decision variables
xC = pulp.LpVariable("Chairs",  lowBound=0, upBound=40)
xT = pulp.LpVariable("Tables",  lowBound=0, upBound=30)
```

```
xS = pulp.LpVariable("Shelves", lowBound=0, upBound=50)

# Objective
model += 45 * xC + 80 * xT + 30 * xS, "Total_Profit"

# Resource constraints
wood_constraint   = 3 * xC + 5 * xT + 2 * xS <= 240
labour_constraint = 2 * xC + 4 * xT + xS      <= 120

model += wood_constraint,   "Wood"
model += labour_constraint, "Labour"

# Solve
model.solve(pulp.PULP_CBC_CMD(msg=0))

print(f"Status  : {pulp.LpStatus[model.status]}")
print(f"Chairs  : {pulp.value(xC):.1f} units/week")
print(f"Tables  : {pulp.value(xT):.1f} units/week")
print(f"Shelves : {pulp.value(xS):.1f} units/week")
print(f"Profit  : ${pulp.value(model.objective):,.2f}/week")
```

```
Status  : Optimal
Chairs  : 35.0 units/week
Tables  : 0.0 units/week
Shelves : 50.0 units/week
Profit  : $3,075.00/week
```

Reading the Solution

The solver returns the **optimal production mix** — the combination that maximises weekly profit without exceeding any resource limit. Notice that the market demand caps on tables and shelves may or may not bind; the shadow prices (below) reveal which constraints are actually limiting growth.

Sensitivity Analysis

An optimal solution answers *what* to do. Sensitivity analysis answers *how confident we should be* in that answer and *where to focus improvement efforts.*

Shadow Prices

The **shadow price** (or **dual value**) of a constraint measures how much the objective would improve if that constraint's right-hand side were relaxed by one unit.

- Shadow price $= 0 \rightarrow$ constraint is **slack** (not binding); relaxing it changes nothing.
- Shadow price $> 0 \rightarrow$ constraint is **binding**; each additional unit of that resource is worth exactly the shadow price in objective improvement.

```
print("Shadow Prices (dual values):")
print(f"  Wood   : ${model.constraints['Wood'].pi:.4f} per board-foot")
print(f"  Labour : ${model.constraints['Labour'].pi:.4f} per hour")

print("\nSlack (unused capacity):")
print(f"  Wood   : {-model.constraints['Wood'].slack:.1f} bf used of 240")
print(f"  Labour : {-model.constraints['Labour'].slack:.1f} hr used of 120")
```

```
Shadow Prices (dual values):
  Wood   : $-0.0000 per board-foot
  Labour : $22.5000 per hour

Slack (unused capacity):
  Wood   : -35.0 bf used of 240
  Labour : 0.0 hr used of 120
```

Reduced Costs

The **reduced cost** of a decision variable is the amount by which its objective coefficient would need to improve before it would enter an optimal solution at a positive level. Variables already in the solution have a reduced cost of zero.

```
print("Reduced Costs:")
for v in [xC, xT, xS]:
    print(f"  {v.name:8s}: {v.dj:.4f}")
```

```
Reduced Costs:
  Chairs  : -0.0000
  Tables  : -10.0000
  Shelves : 7.5000
```

Visualising Resource Utilisation

```
import matplotlib.pyplot as plt

resources  = ["Wood (bf)", "Labour (hr)"]
capacity   = [240, 120]
used       = [
    3 * pulp.value(xC) + 5 * pulp.value(xT) + 2 * pulp.value(xS),
    2 * pulp.value(xC) + 4 * pulp.value(xT) +     pulp.value(xS),
]
```

```
fig, ax = plt.subplots(figsize=(6, 3.5))
x_pos = range(len(resources))

ax.barh(x_pos, capacity, color="#e2e8f0", height=0.5, label="Capacity")
ax.barh(x_pos, used,     color="#2563eb", height=0.5, alpha=0.85, label="Used")

for i, (u, c) in enumerate(zip(used, capacity)):
    ax.text(u + 2, i, f"{u:.0f} / {c}", va="center", fontsize=9)

ax.set_yticks(list(x_pos))
ax.set_yticklabels(resources)
ax.set_xlabel("Units")
ax.set_title("Resource Utilisation at Optimal Solution")
ax.legend(loc="lower right", fontsize=9)
ax.grid(axis="x", alpha=0.3)
plt.tight_layout()
plt.show()
```

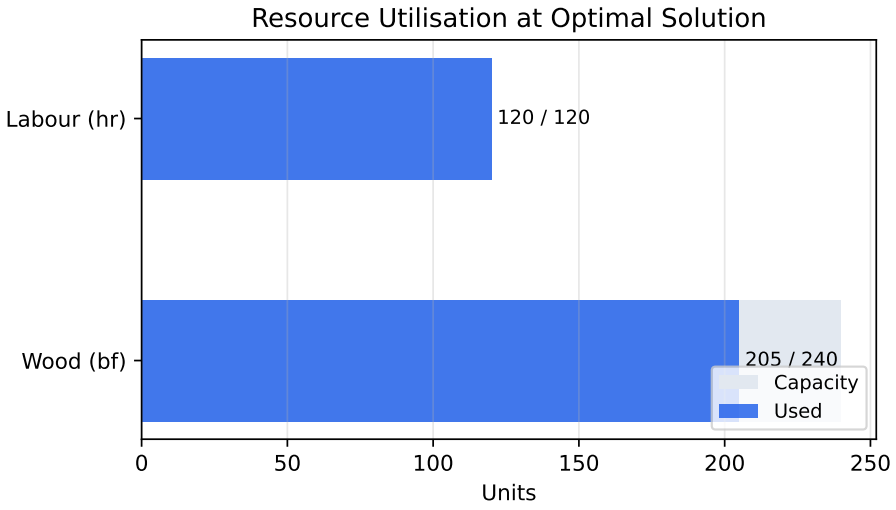


Figure 13: Resource utilisation at the optimal solution

A fully utilised resource (no slack) has a positive shadow price. An underutilised resource has a shadow price of zero — acquiring more of it adds no value at the current optimum.

Duality

Every LP (the **primal**) has a companion problem called the **dual**. The dual offers a different perspective on the same problem — and a powerful tool for understanding what the solution means.

Constructing the Dual

For a primal maximisation problem:

$$\text{Primal:} \quad \max \ \mathbf{c}^T\mathbf{x} \quad \text{s.t. } A\mathbf{x} \leq \mathbf{b},\ \mathbf{x} \geq \mathbf{0}$$

The dual is:

$$\text{Dual:} \quad \min \ \mathbf{b}^T\mathbf{y} \quad \text{s.t. } A^T\mathbf{y} \geq \mathbf{c},\ \mathbf{y} \geq \mathbf{0}$$

Where $\mathbf{y} \in \mathbb{R}^m$ are the **dual variables** — one per primal constraint.

Economic Interpretation

The dual variables y_i are precisely the **shadow prices** from the previous section. They represent the **marginal value** of each resource: the maximum price a rational firm should be willing to pay for one additional unit of resource i.

The dual constraint $A^T\mathbf{y} \geq \mathbf{c}$ says: the implicit value of the resources consumed by each product must be at least as large as that product's profit. If it were less, you could profitably buy more resources and produce more — contradicting optimality.

Strong Duality

The most important result in LP:

At optimality, the primal and dual objectives are equal:

$$\mathbf{c}^T\mathbf{x}^* = \mathbf{b}^T\mathbf{y}^*$$

This means the maximum profit equals the minimum total imputed resource value — every unit of profit can be fully attributed to the resources that produced it.

Special Cases

Infeasibility

A problem is **infeasible** if no point satisfies all constraints simultaneously. This usually signals a modelling error — incompatible constraints, overly tight bounds, or a missing variable. PuLP returns `"Infeasible"` and the objective value is undefined.

```
m = pulp.LpProblem("Infeasible_Demo", pulp.LpMaximize)
x = pulp.LpVariable("x", lowBound=0)
m += x
m += x >= 10
m += x <= 5
m.solve(pulp.PULP_CBC_CMD(msg=0))
print(f"Status: {pulp.LpStatus[m.status]}")  # Infeasible
```

```
Status: Infeasible
```

Unboundedness

A problem is **unbounded** if the objective can be improved without limit. This also signals a model error — a missing constraint that should cap the objective. PuLP returns `"Unbounded"`.

Multiple Optima

When the objective function is parallel to a binding constraint, the optimal objective value is achieved by an entire **edge** of the feasible region — infinitely many solutions all with the same objective value. Solvers return one vertex, but any convex combination of the degenerate vertices is also optimal.

A Larger Example: Diet Optimisation

The **diet problem** — minimise cost while meeting nutritional requirements — is an OR classic. Introduced by Stigler in 1945, it was one of the first practical LPs ever solved.

```
import pulp
import pandas as pd

# Food options: (cost per serving $, protein g, carbs g, fat g, calories)
foods = {
    "Oats":        (0.30, 5,  27, 3,  150),
    "Eggs":        (0.25, 6,   1, 5,   70),
```

```
    "Chicken":     (1.20, 25,  0, 5,  165),
    "Broccoli":    (0.40, 3,   6, 0,   55),
    "Brown Rice":  (0.20, 3,  45, 1,  215),
    "Greek Yogurt":(0.60, 10,  7, 0,   59),
    "Almonds":     (0.50, 6,   6, 14,  164),
}

# Minimum daily requirements
min_protein  = 50   # g
min_carbs    = 130  # g
max_fat      = 65   # g
min_calories = 1800
max_calories = 2200

model = pulp.LpProblem("Diet_Optimisation", pulp.LpMinimize)

servings = {
    food: pulp.LpVariable(food.replace(" ", "_"), lowBound=0)
    for food in foods
}

# Objective: minimise cost
model += pulp.lpSum(foods[f][0] * servings[f] for f in foods), "Total_Cost"

# Nutritional constraints
model += pulp.lpSum(foods[f][1] * servings[f] for f in foods) >= min_protein,  "Protein
model += pulp.lpSum(foods[f][2] * servings[f] for f in foods) >= min_carbs,    "Carbs"
model += pulp.lpSum(foods[f][3] * servings[f] for f in foods) <= max_fat,      "Fat"
model += pulp.lpSum(foods[f][4] * servings[f] for f in foods) >= min_calories, "Min_Ca
model += pulp.lpSum(foods[f][4] * servings[f] for f in foods) <= max_calories, "Max_Ca

model.solve(pulp.PULP_CBC_CMD(msg=0))

print(f"Status       : {pulp.LpStatus[model.status]}")
print(f"Daily cost   : ${pulp.value(model.objective):.2f}\n")
print("Optimal servings:")
for food, var in servings.items():
    qty = pulp.value(var)
    if qty and qty > 0.01:
        print(f"  {food:15s}: {qty:.2f} serving(s)")

# Nutritional totals
protein  = sum(foods[f][1] * pulp.value(servings[f]) for f in foods)
carbs    = sum(foods[f][2] * pulp.value(servings[f]) for f in foods)
fat      = sum(foods[f][3] * pulp.value(servings[f]) for f in foods)
```

```
calories = sum(foods[f][4] * pulp.value(servings[f]) for f in foods)

print(f"\nNutritional totals:")
print(f"  Protein  : {protein:.1f}g  (min {min_protein}g)")
print(f"  Carbs    : {carbs:.1f}g  (min {min_carbs}g)")
print(f"  Fat      : {fat:.1f}g  (max {max_fat}g)")
print(f"  Calories : {calories:.0f} kcal  ({min_calories}-{max_calories})")
```

```
Status       : Optimal
Daily cost   : $2.59

Optimal servings:
  Eggs           : 4.95 serving(s)
  Brown Rice     : 6.76 serving(s)

Nutritional totals:
  Protein  : 50.0g  (min 50g)
  Carbs    : 309.1g  (min 130g)
  Fat      : 31.5g  (max 65g)
  Calories : 1800 kcal  (1800-2200)
```

The diet problem highlights a common LP characteristic: the optimal solution is often **corner-heavy**, concentrating on a small number of foods. This is mathematically correct but practically unappealing. In practice, additional constraints (variety limits, maximum servings per food) or objective modifications introduce more realistic solutions.

When LP Is and Is Not Appropriate

LP is the right tool when:

- The objective and all constraints are genuinely linear
- Decision variables can take any non-negative real value (fractional solutions are acceptable)
- The problem is deterministic — parameters are known

LP is the **wrong** tool when:

Situation	Better approach
Decisions must be whole numbers (order 5 trucks, not 4.7)	Integer Programming (Chapter 3)
Objective or constraints are nonlinear (quadratic cost, diminishing returns)	Nonlinear Programming (Chapter 6)

Situation	Better approach
Parameters are uncertain (demand varies stochastically)	Stochastic Programming (Chapter 5)
The problem involves sequential decisions over time	Dynamic Programming / RL (Chapter 9)

Recognising which tool fits which problem is the core skill of an OR practitioner.

Chapter Summary

- Linear Programming optimises a linear objective subject to linear constraints — the foundational model in Operations Research
- The feasible region is a convex polyhedron; optimal solutions always occur at vertices
- The **Simplex Method** searches vertices efficiently by moving along improving edges
- **Shadow prices** quantify the marginal value of each constrained resource and guide investment decisions
- **Duality** links every maximisation problem to a companion minimisation problem; at optimality, their objectives are equal
- PuLP provides a clean Python interface for formulating, solving, and interrogating LP models via the CBC solver
- LP is appropriate when the model is linear and decision variables are continuous; integrality, nonlinearity, or uncertainty require different methods

Integer Programming

Chapter 3 – Part II: Core Optimization Techniques

Why Integers Matter

Linear Programming assumes decision variables can take any non-negative real value. That assumption fails the moment a decision becomes inherently discrete: you cannot hire 3.7 employees, lease 2.4 trucks, or open a fraction of a warehouse.

Integer Programming (IP) extends LP by requiring some or all decision variables to take integer values. The addition of one word — *integer* — transforms the problem structurally. The continuous relaxation's elegant geometry breaks down: the feasible region is no longer a convex polyhedron but a finite set of discrete points, and the Simplex Method cannot navigate it directly.

Three variants cover most practical problems:

Variant	Variables	Typical Use
Pure Integer Program (ILP)	All integer	Facility location, crew scheduling
Mixed-Integer Program (MIP)	Some integer, some continuous	Production planning with setup costs
Binary Integer Program (BIP)	All 0/1	Selection, assignment, routing

Binary variables are the most expressive building block in IP. An enormous range of logical conditions — if/then, either/or, at-most-K — can be encoded as linear constraints on binary variables.

Standard Form

A general Mixed-Integer Program:

$$\text{Maximize} \quad \mathbf{c}^T\mathbf{x} + \mathbf{d}^T\mathbf{y}$$

$$\text{subject to:} \quad A\mathbf{x} + B\mathbf{y} \leq \mathbf{b}$$

$$\mathbf{x} \geq \mathbf{0}, \quad \mathbf{y} \in \mathbb{Z}^p_{\geq 0}$$

Where $\mathbf{x}$ are continuous variables and $\mathbf{y}$ are integer variables. When all variables are binary ($\mathbf{y} \in \{0,1\}^p$), the problem is a **Binary IP**.

Binary Variables as Logic

Binary variables encode decisions: $y = 1$ means "yes", $y = 0$ means "no". Their real power is in representing logical relationships as linear constraints.

Either/Or Constraints

If at least one of two constraints must hold:

$$a_1^T\mathbf{x} \leq b_1 + M(1-y)$$
$$a_2^T\mathbf{x} \leq b_2 + My$$

When $y = 0$, constraint 1 is enforced and constraint 2 is relaxed (by the big-M). When $y = 1$, the reverse. This is the **Big-M method** — choose M large enough to make the relaxed constraint non-binding, but as small as possible to avoid numerical issues.

If-Then Constraints

"If project i is selected, project j must also be selected":

$$y_j \geq y_i$$

"If facility i is open, at least 10 units must be assigned to it":

$$\sum_j x_{ij} \geq 10 \cdot y_i$$

At-Most-K Selection

"Select at most K items from a set of n candidates":

$$\sum_{i=1}^{n} y_i \leq K$$

These patterns compose. Complex combinatorial constraints — project dependencies, scheduling windows, capacity thresholds — reduce to linear inequalities over binary variables.

Branch and Bound

The standard algorithm for solving IPs is **Branch and Bound**. It exploits a key observation: the LP relaxation (ignoring integrality) provides an upper bound on the IP objective. If the relaxation's solution happens to be integer, it is also optimal for the IP.

The Algorithm

1. **Relax** — solve the LP relaxation to get an upper bound $\bar{Z}$.
2. **Branch** — if the relaxation solution has a fractional variable $y_j = 0.6$, create two subproblems: one with $y_j \leq 0$ and one with $y_j \geq 1$.
3. **Bound** — solve each subproblem's relaxation. Subproblems whose relaxation bound is worse than the current best integer solution are **pruned**.
4. **Repeat** — continue branching until all subproblems are solved or pruned.

The result is a **search tree**. Branch and Bound intelligently prunes branches that cannot improve on the best integer solution found so far (**incumbent**).

Why It Works in Practice

In theory, the search tree can be exponential in the number of integer variables. In practice, modern solvers combine Branch and Bound with:

- **Cutting planes** — additional constraints that cut off fractional solutions without removing any integer feasible points, tightening the relaxation
- **Presolving** — simplifying the problem before solving
- **Heuristics** — finding good incumbent solutions early to prune aggressively

Commercial solvers (Gurobi, CPLEX) and open-source solvers (CBC, HiGHS) can solve problems with millions of variables using these techniques.

Python Implementation with PuLP

PuLP handles integer and binary variables with a single argument change:

```
# Continuous (LP)
x = pulp.LpVariable("x", lowBound=0)

# Integer (IP)
x = pulp.LpVariable("x", lowBound=0, cat="Integer")

# Binary (BIP)
x = pulp.LpVariable("x", cat="Binary")
```

The solver (CBC by default) automatically applies Branch and Bound.

Case Study 1: Capital Budgeting

A firm has **$120,000** to invest across six projects. Each project requires full funding (no partial investment) and returns a net present value (NPV).

Project	Cost (k)|NPV(k)	Dependency	
A	30	40	—
B	25	35	—
C	40	55	Requires A
D	20	28	—
E	35	50	Requires A or B
F	15	18	—

Project C can only be funded if A is funded. Project E requires either A or B.

```
import pulp

model = pulp.LpProblem("Capital_Budgeting", pulp.LpMaximize)

projects = ["A", "B", "C", "D", "E", "F"]
cost = {"A": 30, "B": 25, "C": 40, "D": 20, "E": 35, "F": 15}
npv  = {"A": 40, "B": 35, "C": 55, "D": 28, "E": 50, "F": 18}

y = {p: pulp.LpVariable(f"y_{p}", cat="Binary") for p in projects}

# Objective: maximise NPV
model += pulp.lpSum(npv[p] * y[p] for p in projects), "Total_NPV"
```

```
# Budget constraint
model += pulp.lpSum(cost[p] * y[p] for p in projects) <= 120, "Budget"

# C requires A
model += y["C"] <= y["A"], "C_requires_A"

# E requires A or B  (y_E <= y_A + y_B)
model += y["E"] <= y["A"] + y["B"], "E_requires_A_or_B"

model.solve(pulp.PULP_CBC_CMD(msg=0))

print(f"Status      : {pulp.LpStatus[model.status]}")
print(f"Total NPV   : ${pulp.value(model.objective):.0f}k")
total_cost = sum(cost[p] * pulp.value(y[p]) for p in projects)
print(f"Total Cost  : ${total_cost:.0f}k of $120k budget\n")
print("Selected projects:")
for p in projects:
    if pulp.value(y[p]) > 0.5:
        print(f"  Project {p}: cost=${cost[p]}k, NPV=${npv[p]}k")
```

```
Status      : Optimal
Total NPV   : $163k
Total Cost  : $120k of $120k budget

Selected projects:
  Project A: cost=$30k, NPV=$40k
  Project C: cost=$40k, NPV=$55k
  Project E: cost=$35k, NPV=$50k
  Project F: cost=$15k, NPV=$18k
```

Interpreting the Result

The binary formulation captures the all-or-nothing nature of project investment and the dependency logic exactly. An LP relaxation would produce fractional selections (e.g., "fund 60% of project E") that are not actionable. The IP guarantees an implementable decision.

Case Study 2: Facility Location

A retailer is deciding which of five potential warehouse locations to open to serve eight customer zones. Opening a warehouse has a fixed cost. Serving a customer zone from a warehouse has a variable cost per unit.

Decision: - $y_j \in \{0, 1\}$: open warehouse j - $x_{ij} \geq 0$: fraction of zone i's demand

served by warehouse j

Objective: minimise total fixed + variable cost.

```
import pulp
import numpy as np

np.random.seed(42)

n_zones      = 8
n_warehouses = 5

# Fixed costs to open each warehouse ($k)
fixed_cost = [120, 95, 110, 85, 130]

# Variable cost: cost[i][j] = cost per unit to serve zone i from warehouse j
var_cost = np.random.randint(5, 25, size=(n_zones, n_warehouses)).tolist()

# Demand at each zone (units)
demand = [100, 80, 150, 60, 120, 90, 70, 110]

model = pulp.LpProblem("Facility_Location", pulp.LpMinimize)

# Variables
y = [pulp.LpVariable(f"open_{j}", cat="Binary") for j in range(n_warehouses)]
x = [[pulp.LpVariable(f"serve_{i}_{j}", lowBound=0)
      for j in range(n_warehouses)]
     for i in range(n_zones)]

# Objective
fixed_terms    = pulp.lpSum(fixed_cost[j] * y[j] for j in range(n_warehouses))
variable_terms = pulp.lpSum(
    var_cost[i][j] * x[i][j]
    for i in range(n_zones) for j in range(n_warehouses)
)
model += fixed_terms + variable_terms, "Total_Cost"

# Each zone's demand must be fully served
for i in range(n_zones):
    model += pulp.lpSum(x[i][j] for j in range(n_warehouses)) == demand[i], f"Demand_{i

# Can only serve from open warehouses
for i in range(n_zones):
    for j in range(n_warehouses):
        model += x[i][j] <= demand[i] * y[j], f"Link_{i}_{j}"
```

```
model.solve(pulp.PULP_CBC_CMD(msg=0))

print(f"Status         : {pulp.LpStatus[model.status]}")
print(f"Total cost     : ${pulp.value(model.objective):,.0f}k\n")

open_wh = [j for j in range(n_warehouses) if pulp.value(y[j]) > 0.5]
print(f"Open warehouses: {[j+1 for j in open_wh]}")
for j in open_wh:
    served = [i+1 for i in range(n_zones) if pulp.value(x[i][j]) > 0.01]
    print(f"  Warehouse {j+1} serves zones: {served}")
```

```
Status         : Optimal
Total cost     : $6,975k

Open warehouses: [1, 3, 4, 5]
  Warehouse 1 serves zones: [1, 4, 5]
  Warehouse 3 serves zones: [3, 6]
  Warehouse 4 serves zones: [8]
  Warehouse 5 serves zones: [2, 7]
```

The facility location problem is a classic **MIP**: binary variables for the open/close decision, continuous variables for allocation. The linking constraint `x[i][j] <= demand[i] * y[j]` enforces that no zone can be served by a closed warehouse.

The Travelling Salesman Problem

The **Travelling Salesman Problem (TSP)** is the most famous combinatorial optimisation problem: given n cities and pairwise distances, find the shortest tour visiting every city exactly once and returning to the start.

TSP is **NP-hard** — no polynomial-time exact algorithm is known for the general case. Yet it is solvable in practice for hundreds or thousands of cities using IP with specialised cutting planes (Subtour Elimination Constraints).

Formulation

Let $x_{ij} \in \{0, 1\}$: 1 if the tour goes directly from city i to city j.

$$\text{Minimize} \quad \sum_{i \neq j} d_{ij} x_{ij}$$

$$\text{s.t.} \quad \sum_{j \neq i} x_{ij} = 1 \quad \forall i \quad \text{(leave each city once)}$$

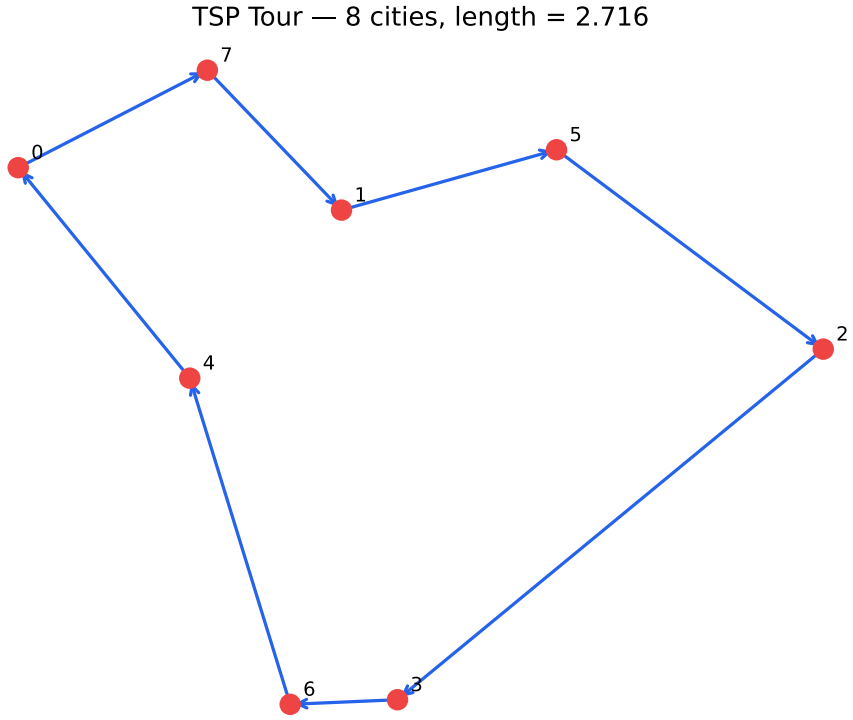
$$\sum_{i \neq j} x_{ij} = 1 \quad \forall j \quad \text{(enter each city once)}$$

Plus **subtour elimination constraints** to prevent disconnected cycles.

```
import pulp
import itertools
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(7)
n = 8
coords = np.random.rand(n, 2)

# Distance matrix
dist = {(i, j): np.linalg.norm(coords[i] - coords[j])
        for i in range(n) for j in range(n) if i != j}

model = pulp.LpProblem("TSP", pulp.LpMinimize)

x = {(i, j): pulp.LpVariable(f"x_{i}_{j}", cat="Binary")
     for i in range(n) for j in range(n) if i != j}

# Objective
model += pulp.lpSum(dist[i, j] * x[i, j] for i, j in x), "Distance"

# Enter and leave each city exactly once
for i in range(n):
    model += pulp.lpSum(x[i, j] for j in range(n) if j != i) == 1, f"Leave_{i}"
    model += pulp.lpSum(x[j, i] for j in range(n) if j != i) == 1, f"Enter_{i}"

# Subtour elimination (Miller-Tucker-Zemlin formulation)
u = {i: pulp.LpVariable(f"u_{i}", lowBound=1, upBound=n - 1, cat="Continuous")
     for i in range(1, n)}

for i in range(1, n):
    for j in range(1, n):
        if i != j:
            model += u[i] - u[j] + (n - 1) * x[i, j] <= n - 2, f"MTZ_{i}_{j}"

model.solve(pulp.PULP_CBC_CMD(msg=0))

# Extract tour
tour = [0]
current = 0
```

```
for _ in range(n - 1):
    nxt = next(j for j in range(n) if j != current and pulp.value(x[current, j]) > 0.5)
    tour.append(nxt)
    current = nxt
tour.append(0)

total_dist = sum(dist[tour[k], tour[k+1]] for k in range(n))
print(f"Tour   : {' → '.join(str(c) for c in tour)}")
print(f"Length : {total_dist:.4f}")

# Plot
fig, ax = plt.subplots(figsize=(6, 5))
for k in range(len(tour) - 1):
    a, b = tour[k], tour[k + 1]
    ax.annotate("", xy=coords[b], xytext=coords[a],
                arrowprops=dict(arrowstyle="->", color="#2563eb", lw=1.5))
ax.scatter(coords[:, 0], coords[:, 1], s=80, color="#ef4444", zorder=3)
for i in range(n):
    ax.annotate(str(i), coords[i], textcoords="offset points",
                xytext=(6, 4), fontsize=9)
ax.set_title(f"TSP Tour - {n} cities, length = {total_dist:.3f}")
ax.axis("off")
plt.tight_layout()
plt.show()
```

```
Tour   : 0 → 7 → 1 → 5 → 2 → 3 → 6 → 4 → 0
Length : 2.7163
```



For larger TSP instances, exact IP approaches (with lazy subtour elimination) or meta-heuristics (simulated annealing, genetic algorithms) are preferred. The Google OR-Tools library provides state-of-the-art TSP solvers.

Common Modelling Patterns

Goal	Binary Formulation
Select exactly k items	$\sum y_i = k$
Select at most k items	$\sum y_i \leq k$
A requires B	$y_A \leq y_B$
At most one of A, B	$y_A + y_B \leq 1$
If $x > 0$ then $y = 1$	$x \leq M \cdot y$
Fixed charge (pay f to use resource)	$f \cdot y + c \cdot x$ in objective; $x \leq M \cdot y$

Mastering these patterns — and combining them — is the core craft of IP modelling.

Practical Guidance

Start with the LP relaxation. Solve without integrality and examine the fractional variables. This reveals the structure of the problem and gives an optimality bound.

Tighten the formulation. Weak formulations (loose bounds on M, redundant constraints) slow the solver by producing poor relaxation bounds. Better formulations lead to tighter LP bounds and fewer Branch and Bound nodes.

Limit solve time for large problems. Real-world MIPs may not solve to provable optimality in reasonable time. Set a time limit and accept the best solution found, together with its **optimality gap** (how far the best integer solution is from the LP bound).

```
# Accept solution within 1% of optimal, or after 60 seconds
model.solve(pulp.PULP_CBC_CMD(msg=0, gapRel=0.01, timeLimit=60))
```

Chapter Summary

- Integer Programming adds integrality requirements to LP, capturing decisions that are inherently discrete
- Binary variables are the primary modelling tool: they encode yes/no choices and represent logical relationships (dependencies, exclusivity, fixed charges) as linear constraints
- **Branch and Bound** solves IPs by systematically exploring a tree of LP relaxations, pruning branches that cannot improve the current best solution
- Modern solvers combine Branch and Bound with cutting planes, presolving, and heuristics to handle large-scale MIPs efficiently
- PuLP exposes integer and binary variables via `cat="Integer"` and `cat="Binary"` with no change to the modelling interface
- Common patterns — selection, dependency, fixed charge — compose into powerful formulations for facility location, scheduling, routing, and capital allocation

Network Optimization

Chapter 4 – Part II: Core Optimization Techniques

Networks Are Everywhere

A network is a set of **nodes** connected by **edges**. That simple structure models an enormous range of real-world systems:

- Road and rail networks — nodes are intersections, edges are roads
- Supply chains — nodes are factories, warehouses, and retailers; edges are shipping lanes
- Telecommunications — nodes are routers, edges are fibre links
- Social networks — nodes are people, edges are relationships
- Project schedules — nodes are tasks, edges are dependencies

Network Optimization exploits the structure of these problems to solve them far more efficiently than general-purpose LP or IP methods. The two key tools are **graph theory** (representing structure) and **network flow** (optimising over that structure).

Python's `networkx` library provides the data structures; `pulp` provides the optimisation. Together they handle problems from shortest paths to minimum cost flows.

Graph Fundamentals

A **graph** $G = (V, E)$ consists of:

- V — a set of **vertices** (nodes)
- E — a set of **edges** (arcs), where each edge connects two vertices

Directed graphs (digraphs) have edges with a direction — from a *tail* node to a *head* node. Flow on an arc moves in one direction only.

Undirected graphs have edges with no direction — you can traverse them either way.

Key properties:

Term	Definition
Degree	Number of edges incident to a node
Path	Sequence of edges connecting two nodes without repeating nodes
Cycle	A path that starts and ends at the same node
Connected	Every pair of nodes has at least one path between them
Tree	A connected graph with no cycles ($\|E\| = \|V\| - 1$)
Spanning tree	A tree that includes all nodes of the graph

```python
import networkx as nx
import matplotlib.pyplot as plt

G = nx.DiGraph()
edges = [
    ("A", "B", 4), ("A", "C", 2), ("B", "C", 5),
    ("B", "D", 10), ("C", "E", 3), ("D", "F", 11),
    ("E", "D", 4), ("E", "F", 7), ("F", "G", 2)
]
G.add_weighted_edges_from(edges)

pos = nx.spring_layout(G, seed=42)
weights = nx.get_edge_attributes(G, "weight")

fig, ax = plt.subplots(figsize=(7, 4))
nx.draw_networkx(G, pos, ax=ax, node_color="#6366f1", node_size=600,
                 font_color="white", font_size=10, arrows=True,
                 arrowsize=15, edge_color="#94a3b8", width=1.5)
nx.draw_networkx_edge_labels(G, pos, edge_labels=weights, font_size=8, ax=ax)
ax.set_title("Directed Weighted Graph")
ax.axis("off")
plt.tight_layout()
plt.show()
```

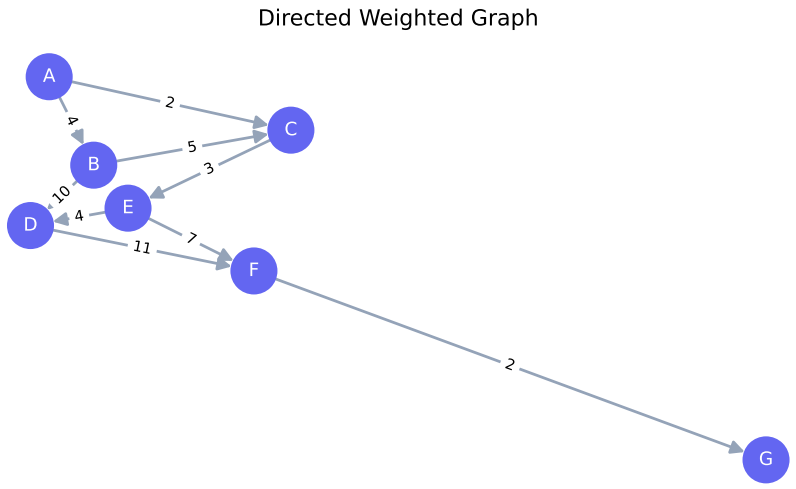


Figure 14: A directed weighted graph

Shortest Path

Problem: Find the minimum-cost path from a source node s to a destination node t in a weighted graph.

Applications: GPS routing, network packet delivery, project critical path.

Dijkstra's Algorithm

For graphs with non-negative edge weights, **Dijkstra's algorithm** finds shortest paths from a single source in $O((V+E)\log V)$ time. It maintains a priority queue of nodes ordered by their current shortest distance from the source, greedily expanding the closest unvisited node at each step.

```
import networkx as nx

G = nx.DiGraph()
edges = [
    ("A", "B", 4), ("A", "C", 2), ("B", "C", 5),
    ("B", "D", 10), ("C", "E", 3), ("D", "F", 11),
    ("E", "D", 4), ("E", "F", 7), ("F", "G", 2)
]
G.add_weighted_edges_from(edges)

path   = nx.dijkstra_path(G, "A", "G", weight="weight")
```

```
length = nx.dijkstra_path_length(G, "A", "G", weight="weight")

print(f"Shortest path : {' → '.join(path)}")
print(f"Total cost    : {length}")
```

```
Shortest path : A → C → E → F → G
Total cost    : 14
```

```
import matplotlib.pyplot as plt

pos = {"A": (0,1), "B": (1,2), "C": (1,0), "D": (2,2),
       "E": (2,0), "F": (3,1), "G": (4,1)}

path_edges = list(zip(path[:-1], path[1:]))
non_path   = [(u, v) for u, v, _ in edges if (u, v) not in path_edges]
weights    = nx.get_edge_attributes(G, "weight")

fig, ax = plt.subplots(figsize=(8, 4))
nx.draw_networkx_nodes(G, pos, node_color="#6366f1", node_size=600, ax=ax)
nx.draw_networkx_labels(G, pos, font_color="white", font_size=10, ax=ax)
nx.draw_networkx_edges(G, pos, edgelist=non_path, edge_color="#cbd5e1",
                       arrows=True, arrowsize=15, ax=ax)
nx.draw_networkx_edges(G, pos, edgelist=path_edges, edge_color="#ef4444",
                       width=2.5, arrows=True, arrowsize=18, ax=ax)
nx.draw_networkx_edge_labels(G, pos, edge_labels=weights, font_size=8, ax=ax)
ax.set_title(f"Shortest Path A → G (cost = {length})")
ax.axis("off")
plt.tight_layout()
plt.show()
```

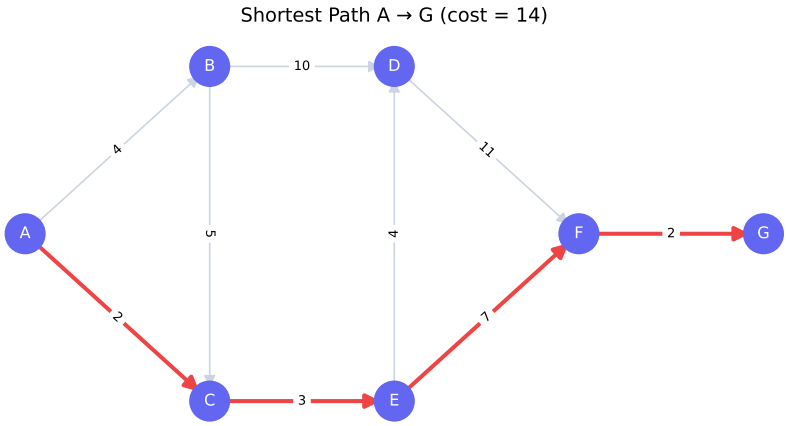


Figure 15: Shortest path from A to G highlighted in red

Bellman-Ford for Negative Weights

Dijkstra fails on graphs with negative edge weights. **Bellman-Ford** handles them in $O(VE)$ time and also detects **negative cycles** (cycles where the total weight is negative — the "shortest path" would be infinitely short by cycling forever).

```
G_neg = nx.DiGraph()
G_neg.add_weighted_edges_from([
    ("S", "A", 6), ("S", "B", 7), ("A", "B", 8), ("A", "C", -4),
    ("B", "C", -3), ("B", "D", 9), ("C", "D", 7), ("D", "A", 2)
])

try:
    path_bf   = nx.bellman_ford_path(G_neg, "S", "D", weight="weight")
    length_bf = nx.bellman_ford_path_length(G_neg, "S", "D", weight="weight")
    print(f"Bellman-Ford path   : {' → '.join(path_bf)}")
    print(f"Total cost          : {length_bf}")
except nx.exception.NetworkXUnbounded:
    print("Negative cycle detected - shortest path is undefined")
```

```
Bellman-Ford path   : S → A → C → D
Total cost          : 9
```

Minimum Spanning Tree

Problem: Connect all nodes in an undirected graph with the minimum total edge weight, using exactly $|V| - 1$ edges (a tree).

Applications: Network cable layout, cluster analysis, approximation algorithms for TSP.

Two classic algorithms:

- **Kruskal's** — sort edges by weight; add the cheapest edge that does not create a cycle
- **Prim's** — grow a tree from a seed node by repeatedly adding the cheapest edge connecting the tree to a new node

Both run in $O(E \log E)$ for typical implementations.

```
import networkx as nx
import matplotlib.pyplot as plt

G_und = nx.Graph()
G_und.add_weighted_edges_from([
    (1,2,4),(1,3,8),(2,3,11),(2,4,8),(3,5,7),(4,5,2),
    (4,6,4),(4,7,9),(5,7,14),(6,7,10),(6,8,2),(7,8,1)
])

mst      = nx.minimum_spanning_tree(G_und, weight="weight")
mst_cost = sum(d["weight"] for _, _, d in mst.edges(data=True))

pos       = nx.spring_layout(G_und, seed=0)
mst_edges = list(mst.edges())
non_mst   = [(u, v) for u, v in G_und.edges() if (u,v) not in mst_edges and (v,u) not i
all_weights = nx.get_edge_attributes(G_und, "weight")

fig, ax = plt.subplots(figsize=(7, 5))
nx.draw_networkx_nodes(G_und, pos, node_color="#6366f1", node_size=500, ax=ax)
nx.draw_networkx_labels(G_und, pos, font_color="white", font_size=10, ax=ax)
nx.draw_networkx_edges(G_und, pos, edgelist=non_mst, edge_color="#cbd5e1", ax=ax)
nx.draw_networkx_edges(G_und, pos, edgelist=mst_edges,
                       edge_color="#ef4444", width=2.5, ax=ax)
nx.draw_networkx_edge_labels(G_und, pos, edge_labels=all_weights, font_size=8, ax=ax)
ax.set_title(f"Minimum Spanning Tree (total cost = {mst_cost})")
ax.axis("off")
plt.tight_layout()
plt.show()
```

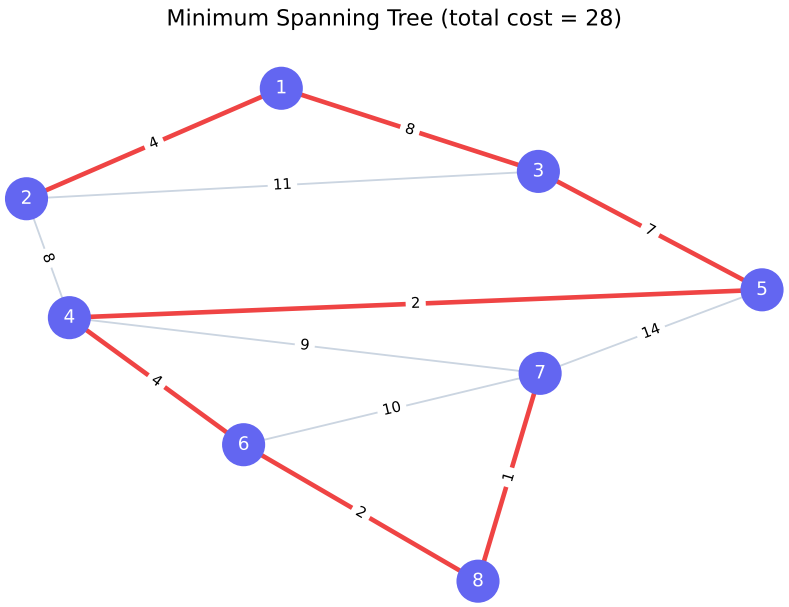


Figure 16: Minimum Spanning Tree highlighted in red

Maximum Flow

Problem: Given a directed graph with edge capacities, find the maximum amount of flow that can be sent from a **source** s to a **sink** t without exceeding any edge capacity.

Applications: Pipeline capacity, internet bandwidth allocation, bipartite matching.

The Max-Flow Min-Cut Theorem

The maximum flow from s to t equals the minimum **cut** — the minimum total capacity of edges that, if removed, would disconnect s from t.

This duality is one of the most elegant results in combinatorial optimisation. It says that the bottleneck in a network is always a cut — a set of edges that, collectively, limits all paths from source to sink.

```
import networkx as nx

G_flow = nx.DiGraph()
```

```
G_flow.add_edge("s", "a", capacity=15)
G_flow.add_edge("s", "b", capacity=4)
G_flow.add_edge("a", "b", capacity=12)
G_flow.add_edge("a", "c", capacity=10)
G_flow.add_edge("b", "d", capacity=10)
G_flow.add_edge("c", "d", capacity=9)
G_flow.add_edge("c", "t", capacity=15)
G_flow.add_edge("d", "t", capacity=10)

flow_value, flow_dict = nx.maximum_flow(G_flow, "s", "t")

print(f"Maximum flow: {flow_value}")
print("\nFlow on each edge:")
for u in flow_dict:
    for v, f in flow_dict[u].items():
        if f > 0:
            cap = G_flow[u][v]["capacity"]
            print(f"  {u} → {v}: {f} / {cap}")
```

```
Maximum flow: 19

Flow on each edge:
  s → a: 15 / 15
  s → b: 4 / 4
  a → b: 5 / 12
  a → c: 10 / 10
  b → d: 9 / 10
  c → t: 10 / 15
  d → t: 9 / 10
```

Visualising Flow

```
import matplotlib.pyplot as plt

pos = {"s": (0,1), "a": (1,2), "b": (1,0), "c": (2,2), "d": (2,0), "t": (3,1)}

edge_labels = {
    (u, v): f"{flow_dict[u].get(v, 0)}/{G_flow[u][v]['capacity']}"
    for u, v in G_flow.edges()
}
edge_colors = [
    "#ef4444" if flow_dict[u].get(v, 0) == G_flow[u][v]["capacity"] else "#94a3b8"
    for u, v in G_flow.edges()
]
```

```
fig, ax = plt.subplots(figsize=(7, 4))
nx.draw_networkx(G_flow, pos, ax=ax, node_color="#6366f1", node_size=600,
                 font_color="white", font_size=10, arrows=True,
                 arrowsize=15, edge_color=edge_colors, width=2)
nx.draw_networkx_edge_labels(G_flow, pos, edge_labels=edge_labels,
                             font_size=8, ax=ax)
ax.set_title(f"Max Flow = {flow_value}  (red = saturated edges)")
ax.axis("off")
plt.tight_layout()
plt.show()
```

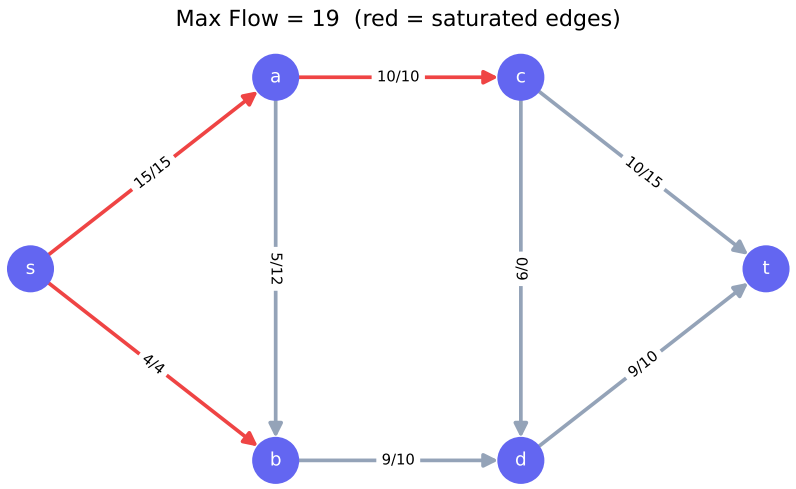


Figure 17: Maximum flow network — edge labels show flow/capacity

Saturated edges (flow = capacity, shown in red) form the **minimum cut** — these are the bottlenecks limiting total throughput.

Minimum Cost Flow

Minimum Cost Flow (MCF) is the most general network flow problem. It subsumes shortest path, max flow, transportation, and assignment problems as special cases.

Formulation: Each arc (i, j) has a **capacity** u_{ij}, a **lower bound** l_{ij}, and a **unit cost** c_{ij}. Each node has a **supply** b_i (positive if it generates flow, negative if it consumes flow, zero if it is a transshipment node).

$$\text{Minimize} \quad \sum_{(i,j)\in E} c_{ij} x_{ij}$$

$$\text{s.t.} \quad \sum_{j:(i,j)\in E} x_{ij} - \sum_{j:(j,i)\in E} x_{ji} = b_i \quad \forall i \in V \quad \text{(flow balance)}$$

$$l_{ij} \le x_{ij} \le u_{ij} \quad \forall (i,j) \in E$$

The flow balance constraint says: flow out minus flow in equals supply at each node. Supply nodes ($b_i > 0$) generate flow; demand nodes ($b_i < 0$) absorb it.

```
import pulp

# Three suppliers (S1, S2, S3), three customers (C1, C2, C3), two warehouses (W1, W2)
# Cost per unit on each arc, capacity, supply/demand

nodes = ["S1", "S2", "S3", "W1", "W2", "C1", "C2", "C3"]

supply = {"S1": 120, "S2": 80,  "S3": 100,
          "W1": 0,   "W2": 0,
          "C1": -90, "C2": -110, "C3": -100}

arcs = {
    ("S1", "W1"): (8,  120), ("S1", "W2"): (6,  120),
    ("S2", "W1"): (5,  80),  ("S2", "W2"): (9,  80),
    ("S3", "W1"): (7,  100), ("S3", "W2"): (4,  100),
    ("W1", "C1"): (3,  200), ("W1", "C2"): (5,  200), ("W1", "C3"): (6,  200),
    ("W2", "C1"): (7,  200), ("W2", "C2"): (2,  200), ("W2", "C3"): (4,  200),
}  # (cost, capacity)

model = pulp.LpProblem("Min_Cost_Flow", pulp.LpMinimize)

x = {(i, j): pulp.LpVariable(f"x_{i}_{j}", lowBound=0, upBound=cap)
     for (i, j), (_, cap) in arcs.items()}

# Objective
model += pulp.lpSum(cost * x[i, j] for (i, j), (cost, _) in arcs.items())

# Flow balance
for node in nodes:
    out_flow = pulp.lpSum(x[i, j] for (i, j) in arcs if i == node)
    in_flow  = pulp.lpSum(x[i, j] for (i, j) in arcs if j == node)
    model += out_flow - in_flow == supply[node], f"Balance_{node}"

model.solve(pulp.PULP_CBC_CMD(msg=0))
```

```
print(f"Status      : {pulp.LpStatus[model.status]}")
print(f"Total cost  : ${pulp.value(model.objective):,.0f}\n")
print("Flow routing:")
for (i, j), var in x.items():
    flow = pulp.value(var)
    if flow and flow > 0.01:
        print(f"  {i} → {j}: {flow:.0f} units")
```

```
Status      : Optimal
Total cost  : $2,430

Flow routing:
  S1 → W1: 10 units
  S1 → W2: 110 units
  S2 → W1: 80 units
  S3 → W2: 100 units
  W1 → C1: 90 units
  W2 → C2: 110 units
  W2 → C3: 100 units
```

MCF is solvable in polynomial time using the **network simplex algorithm** — a specialised version of the Simplex Method that exploits the tree structure of network flow bases. It is typically orders of magnitude faster than general LP for network problems.

The Transportation Problem

The **transportation problem** is a special case of MCF with a bipartite structure: m supply nodes and n demand nodes, all connected directly (no intermediate transshipment nodes).

It models the classic question: *how to ship goods from warehouses to customers at minimum cost, given supply and demand constraints?*

```
import pulp
import numpy as np
import matplotlib.pyplot as plt

# 3 suppliers, 4 customers
supply   = [300, 400, 500]
demand   = [250, 350, 400, 200]
cost_mat = np.array([
    [2, 3, 1, 5],
    [7, 3, 4, 2],
```

```
    [6, 1, 3, 4],
])

n_sup = len(supply)
n_dem = len(demand)

model = pulp.LpProblem("Transportation", pulp.LpMinimize)

x = [[pulp.LpVariable(f"x_{i}_{j}", lowBound=0)
      for j in range(n_dem)] for i in range(n_sup)]

# Objective
model += pulp.lpSum(cost_mat[i, j] * x[i][j]
                    for i in range(n_sup) for j in range(n_dem))

# Supply constraints
for i in range(n_sup):
    model += pulp.lpSum(x[i][j] for j in range(n_dem)) <= supply[i], f"Supply_{i}"

# Demand constraints
for j in range(n_dem):
    model += pulp.lpSum(x[i][j] for i in range(n_sup)) == demand[j], f"Demand_{j}"

model.solve(pulp.PULP_CBC_CMD(msg=0))

print(f"Status     : {pulp.LpStatus[model.status]}")
print(f"Total cost : ${pulp.value(model.objective):,.0f}\n")

print("Shipment plan (units):")
header = "         " + "  ".join(f"Cust {j+1:1d}" for j in range(n_dem))
print(header)
for i in range(n_sup):
    row = f"Supplier {i+1}: " + "  ".join(
        f"{pulp.value(x[i][j]):6.0f}" for j in range(n_dem)
    )
    print(row)
```

```
Status     : Optimal
Total cost : $2,550

Shipment plan (units):
         Cust 1  Cust 2  Cust 3  Cust 4
Supplier 1:    250       0      50       0
Supplier 2:      0       0     200     200
Supplier 3:      0     350     150       0
```

Project Scheduling: CPM

The **Critical Path Method (CPM)** applies shortest/longest path algorithms to project scheduling. Activities are represented as edges (or nodes), with durations as weights. The **critical path** is the longest path through the network — it determines the minimum project duration.

Any delay on the critical path delays the entire project. Activities not on the critical path have **float** (slack) — they can be delayed without affecting the project deadline.

```
import networkx as nx
import matplotlib.pyplot as plt

# Activities: (predecessor, successor, duration)
activities = [
    ("START", "A", 0), ("START", "B", 0), ("START", "C", 0),
    ("A", "D", 3), ("A", "E", 3),
    ("B", "E", 5), ("B", "F", 5),
    ("C", "G", 2),
    ("D", "END", 4), ("E", "END", 6), ("F", "END", 2), ("G", "END", 7)
]

G = nx.DiGraph()
for pred, succ, dur in activities:
    G.add_edge(pred, succ, duration=dur)

# Longest path = critical path
critical_length = nx.dag_longest_path_length(G, weight="duration")
critical_path   = nx.dag_longest_path(G, weight="duration")

print(f"Project duration : {critical_length} days")
print(f"Critical path    : {' → '.join(critical_path)}")

# Early start times via topological sort
topo_order  = list(nx.topological_sort(G))
early_start = {node: 0 for node in G.nodes()}
for node in topo_order:
    for succ in G.successors(node):
        dur = G[node][succ]["duration"]
        early_start[succ] = max(early_start[succ], early_start[node] + dur)

print("\nEarly start times:")
for node, es in sorted(early_start.items(), key=lambda x: x[1]):
```

```
    print(f"  {node:6s}: day {es}")
```

```
Project duration : 11 days
Critical path    : START → B → E → END

Early start times:
  START : day 0
  A     : day 0
  B     : day 0
  C     : day 0
  G     : day 2
  D     : day 3
  E     : day 5
  F     : day 5
  END   : day 11
```

The critical path tells a project manager exactly where to focus resources. Crashing (accelerating) a non-critical activity wastes money; crashing a critical activity directly reduces project duration.

Choosing the Right Algorithm

Problem	Algorithm	Complexity
Single-source shortest path (non-negative weights)	Dijkstra	$O((V+E)\log V)$
Single-source shortest path (negative weights)	Bellman-Ford	$O(VE)$
All-pairs shortest path	Floyd-Warshall	$O(V^3)$
Minimum spanning tree	Kruskal / Prim	$O(E \log E)$
Maximum flow	Ford-Fulkerson / Push-relabel	$O(VE^2)$
Minimum cost flow	Network Simplex	$O(VE \log V \log(VC))$
Transportation	Simplex (network form)	Fast in practice
Bipartite matching	Hungarian algorithm	$O(V^3)$

For most practical problems, `networkx` and `pulp` provide sufficient performance. For large-scale production systems (millions of nodes), Google OR-Tools' network flow solver or Gurobi's built-in network simplex are preferred.

Chapter Summary

- Network Optimization models systems as graphs (nodes and edges) and finds optimal routes, flows, or spanning structures
- **Shortest path** (Dijkstra, Bellman-Ford) solves routing and scheduling problems; the critical path in project management is the longest-path analogue
- **Minimum spanning tree** (Kruskal, Prim) connects all nodes at minimum total edge cost — used in network design and cluster analysis
- **Maximum flow** (Ford-Fulkerson) finds the throughput limit of a network; by the Max-Flow Min-Cut theorem, this equals the minimum cut capacity
- **Minimum cost flow** is the most general network model, subsuming transportation, assignment, and shortest path as special cases
- Python's `networkx` provides graph data structures and classical algorithms; `pulp` solves formulated network problems as LP/MIP
- Network algorithms exploit problem structure to achieve dramatically better performance than general LP solvers on the same problems

Part III: Uncertainty and Data-Driven Approaches

Stochastic Programming

Learning Objectives

- Formulate two-stage stochastic programs with recourse
- Understand scenario-based representations of uncertainty
- Apply Sample Average Approximation (SAA) to data-driven problems
- Quantify the value of modeling uncertainty via VSS and EVPI
- Solve stochastic programs using PuLP and NumPy

Why Deterministic Models Break Down

Every OR model in Part II assumed we know the problem data exactly. In practice, we rarely do. Demand fluctuates. Processing times vary. Costs change. When we ignore this uncertainty and solve a deterministic model using expected values, we often make decisions that perform poorly when reality diverges from the average.

Consider a company planning manufacturing capacity for the next year. Demand could be low, medium, or high depending on market conditions. If the company installs capacity for average demand:

- When demand is high, it leaves revenue on the table (no capacity to fill orders).
- When demand is low, it has idle capacity it paid for.

Neither outcome is optimal. The correct approach is to model the uncertainty explicitly and make decisions that are good *on average across scenarios*, accounting for the ability to adjust operations after uncertainty resolves. This is stochastic programming.

The Newsvendor Problem

The newsvendor problem is the simplest and most instructive stochastic program. It appears throughout supply chain management, revenue management, and inventory control.

Setup. A newsvendor orders q units of a product before observing demand D. Each unit costs c to purchase and sells for $p > c$. Unsold units are salvaged at $s < c$. The vendor must commit to q before demand is known.

Profit as a function of order quantity and realized demand:

$$\pi(q, D) = p \min(q, D) + s \max(q - D, 0) - cq \tag{1}$$

The vendor wants to choose q to maximize expected profit:

$$\max_{q} \ \mathbb{E}_D[\pi(q, D)] \tag{2}$$

Closed-form solution. The expected profit is concave in q, and the optimal order quantity has a clean analytical solution. Taking the derivative of $\mathbb{E}[\pi(q, D)]$ with respect to q and setting it to zero yields the *critical ratio*:

$$q^* = F^{-1}\left(\frac{p - c}{p - s}\right) \tag{3}$$

where F^{-1} is the inverse CDF of demand. The ratio $\frac{p-c}{p-s}$ is the *critical ratio* — the probability of demand falling below the optimal order quantity. Intuitively, order more when margins are high relative to overage cost; order less when salvage is low.

```
import numpy as np
from scipy import stats
import plotly.graph_objects as go
from plotly.subplots import make_subplots

# Parameters
c = 10    # purchase cost per unit
p = 25    # selling price per unit
s = 4     # salvage value per unit

mu, sigma = 100, 20   # demand ~ Normal(100, 20)

# Critical ratio and optimal q
cr = (p - c) / (p - s)
q_star = stats.norm.ppf(cr, mu, sigma)
```

```
# Simulate expected profit over range of q values
q_range = np.linspace(40, 180, 300)

def expected_profit(q, mu, sigma, c, p, s, n_sim=50_000):
    D = np.random.default_rng(42).normal(mu, sigma, n_sim)
    profit = p * np.minimum(q, D) + s * np.maximum(q - D, 0) - c * q
    return profit.mean()

E_profit = [expected_profit(q, mu, sigma, c, p, s) for q in q_range]

fig = go.Figure()
fig.add_trace(go.Scatter(
    x=q_range, y=E_profit,
    mode='lines', name='E[Profit(q)]',
    line=dict(color='steelblue', width=2)
))
fig.add_vline(
    x=q_star, line_dash='dash', line_color='crimson',
    annotation_text=f"q* = {q_star:.1f}  (CR = {cr:.2f})",
    annotation_position='top right'
)
fig.update_layout(
    xaxis_title='Order Quantity q',
    yaxis_title='Expected Profit ($)',
    template='plotly_white',
    height=400
)
fig.show()
```

```
Unable to display output for mime type(s): text/html
```

(a) Expected profit as a function of order quantity for a newsvendor with normal demand. The optimal quantity is at the critical ratio.

```
Unable to display output for mime type(s): text/html
```

(b)

Figure 18

The newsvendor is solvable analytically because it has a single decision and a single uncertain parameter. Real problems have many decisions and many uncertain parameters, and the recourse structure is more complex. For these, we need the general stochastic programming framework.

Two-Stage Stochastic Programming

Most practical stochastic programs follow a **two-stage** structure:

1. **First stage (here-and-now):** Make a decision x *before* uncertainty resolves. This decision is the same regardless of what scenario occurs.
2. **Uncertainty resolves:** A scenario ξ (demand, cost, failure, etc.) is revealed.
3. **Second stage (wait-and-see / recourse):** Make a corrective decision $y(\xi)$ *after* observing ξ. The recourse action can depend on the realized scenario, but it may be costly.

The general two-stage stochastic linear program is:

$$\min_{x} \; c^\top x + \mathbb{E}_\xi[Q(x,\xi)] \tag{4}$$

$$\text{subject to} \quad Ax = b, \quad x \geq 0$$

where $Q(x,\xi)$ is the **recourse function** — the optimal cost of the second-stage problem given first-stage decision x and scenario ξ:

$$Q(x,\xi) = \min_{y} \; q(\xi)^\top y \tag{5}$$

$$\text{subject to} \quad T(\xi)\,x + W\,y = h(\xi), \quad y \geq 0$$

The matrices $T(\xi)$ and $h(\xi)$ are **technology** and **right-hand side** parameters that depend on the scenario. W (the **recourse matrix**) is typically fixed.

> **Note**
>
> **Recourse** means the ability to correct a decision after uncertainty resolves — but at a cost. Stochastic programming assumes you *know* the distribution of ξ when making the first-stage decision, even though you don't yet know the realization.

Scenario-Based Formulation

In practice, the expectation in Equation 4 is approximated by a finite set of **scenarios** $\{\xi^1, \xi^2, \ldots, \xi^S\}$ with probabilities $\{p_1, p_2, \ldots, p_S\}$:

$$\min_{x} \; c^\top x + \sum_{s=1}^{S} p_s\, Q(x,\xi^s) \tag{6}$$

This is the **extensive form** — expanding all scenarios explicitly into one large LP. Substituting the second-stage problem:

$$\min_{x,\{y^s\}} \; c^\top x + \sum_{s=1}^{S} p_s \, q^{s\top} y^s \tag{7}$$

$$\text{subject to} \quad Ax = b, \quad x \geq 0$$

$$T^s x + W y^s = h^s \quad \forall s \in \{1, \ldots, S\}$$

$$y^s \geq 0 \quad \forall s$$

The first-stage variable x appears in *every* scenario constraint — it's the **linking variable** that couples all scenarios. This coupling is what makes stochastic programs different from simply solving S independent LPs.

Worked Example: Capacity Planning Under Demand Uncertainty

A manufacturer must decide how much production capacity x (machine-hours) to install before the demand for the coming year is known. After demand is revealed, the manufacturer can produce up to capacity and incurs a penalty for unmet demand (e.g., lost sales or expedited sourcing).

Parameters: - Capacity cost: \$50 per machine-hour - Production cost: \$20 per unit (one unit = one machine-hour) - Penalty for unmet demand: \$80 per unit - Demand scenarios: Low (80 units), Medium (120 units), High (160 units) - Scenario probabilities: 0.25, 0.50, 0.25

Decision variables: - x: capacity installed (first stage) - y_s: production in scenario s (second stage) - u_s: unmet demand in scenario s (second stage)

Extensive form:

$$\min_{x, y_s, u_s} \; 50x + \sum_{s} p_s (20 y_s + 80 u_s)$$

$$\text{subject to:}$$

$$y_s \leq x \quad \forall s \quad \text{(capacity constraint)}$$

$$y_s + u_s = d_s \quad \forall s \quad \text{(demand balance)}$$

$$x, y_s, u_s \geq 0 \quad \forall s$$

Table 1: Two-stage stochastic program for capacity planning: scenarios, probabilities, and second-stage costs.

```
import pulp
import pandas as pd

# Problem data
c_cap   = 50    # capacity cost ($/unit)
c_prod  = 20    # production cost ($/unit)
c_pen   = 80    # unmet demand penalty ($/unit)

demands = [80, 120, 160]
probs   = [0.25, 0.50, 0.25]
S = len(demands)

# Extensive form LP
prob = pulp.LpProblem("CapacityPlanning_SP", pulp.LpMinimize)

# Variables
x = pulp.LpVariable("capacity", lowBound=0)
y = [pulp.LpVariable(f"production_s{s}", lowBound=0) for s in range(S)]
u = [pulp.LpVariable(f"unmet_s{s}",      lowBound=0) for s in range(S)]

# Objective
prob += (
    c_cap * x
    + pulp.lpSum(probs[s] * (c_prod * y[s] + c_pen * u[s]) for s in range(S))
)

# Constraints
for s in range(S):
    prob += y[s] <= x,           f"capacity_s{s}"
    prob += y[s] + u[s] == demands[s], f"demand_balance_s{s}"

prob.solve(pulp.PULP_CBC_CMD(msg=0))

x_opt = pulp.value(x)
total_cost = pulp.value(prob.objective)

rows = []
for s in range(S):
    rows.append({
        "Scenario": f"s{s+1}",
        "Demand":   demands[s],
        "Prob":     probs[s],
        "Production": round(pulp.value(y[s]), 1),
        "Unmet":    round(pulp.value(u[s]), 1),
        "2nd-Stage Cost": round(probs[s] * (c_prod*pulp.value(y[s]) + c_pen*pulp.value
    })

df = pd.DataFrame(rows)
print(f"Optimal capacity: {x_opt:.1f} units")
print(f"Total expected cost: ${total_cost:.2f}")
print()
print(df.to_string(index=False))
```

```
Optimal capacity: 80.0 units
```

The stochastic solution installs capacity that balances the cost of idle capacity (low demand) against the penalty for unmet demand (high demand). It is neither the low-demand nor the high-demand optimal — it is the best decision *on average across scenarios*.

Value of the Stochastic Solution

How much does it cost to *ignore* uncertainty? We quantify this with two metrics.

Expected Value (EV) solution. Solve the deterministic problem using expected demand $\bar{d} = \sum_s p_s d_s$:

$$\bar{d} = 0.25 \times 80 + 0.50 \times 120 + 0.25 \times 160 = 120$$

Install capacity x^{EV} for 120 units, then evaluate this fixed capacity across all scenarios.

Value of the Stochastic Solution (VSS):

$$\text{VSS} = \text{EEV} - \text{RP} \tag{8}$$

where EEV is the *expected cost of the EV solution* (using x^{EV} and optimizing recourse) and RP is the *recourse problem* (stochastic solution cost). VSS 0 always; larger VSS means more value from modeling uncertainty.

Expected Value of Perfect Information (EVPI):

$$\text{EVPI} = \text{RP} - \text{WS} \tag{9}$$

where WS is the *wait-and-see* cost — if we could observe each scenario before deciding capacity and optimize separately. EVPI is the most we would pay for a perfect forecast.

```
# --- Recourse Problem (already solved above) ---
RP = total_cost

# --- EV solution: solve at expected demand, evaluate across scenarios ---
d_bar = sum(p * d for p, d in zip(probs, demands))   # = 120

prob_ev = pulp.LpProblem("EV", pulp.LpMinimize)
x_ev = pulp.LpVariable("x_ev", lowBound=0)
prob_ev += c_cap * x_ev
prob_ev += x_ev >= 0
prob_ev.solve(pulp.PULP_CBC_CMD(msg=0))

# Optimal EV capacity: solve 1-scenario problem at d_bar
```

```
prob_ev2 = pulp.LpProblem("EV2", pulp.LpMinimize)
xe  = pulp.LpVariable("xe",  lowBound=0)
ye  = pulp.LpVariable("ye",  lowBound=0)
ue  = pulp.LpVariable("ue",  lowBound=0)
prob_ev2 += c_cap * xe + c_prod * ye + c_pen * ue
prob_ev2 += ye <= xe
prob_ev2 += ye + ue == d_bar
prob_ev2.solve(pulp.PULP_CBC_CMD(msg=0))
x_ev_opt = pulp.value(xe)

# EEV: use x_ev_opt, optimize recourse per scenario
EEV = c_cap * x_ev_opt
for s in range(S):
    prob_r = pulp.LpProblem(f"Recourse_s{s}", pulp.LpMinimize)
    yr = pulp.LpVariable("yr", lowBound=0)
    ur = pulp.LpVariable("ur", lowBound=0)
    prob_r += c_prod * yr + c_pen * ur
    prob_r += yr <= x_ev_opt
    prob_r += yr + ur == demands[s]
    prob_r.solve(pulp.PULP_CBC_CMD(msg=0))
    EEV += probs[s] * pulp.value(prob_r.objective)

# --- Wait-and-See: solve each scenario independently ---
WS = 0
for s in range(S):
    prob_ws = pulp.LpProblem(f"WS_s{s}", pulp.LpMinimize)
    xw = pulp.LpVariable("xw", lowBound=0)
    yw = pulp.LpVariable("yw", lowBound=0)
    uw = pulp.LpVariable("uw", lowBound=0)
    prob_ws += c_cap * xw + c_prod * yw + c_pen * uw
    prob_ws += yw <= xw
    prob_ws += yw + uw == demands[s]
    prob_ws.solve(pulp.PULP_CBC_CMD(msg=0))
    WS += probs[s] * pulp.value(prob_ws.objective)

VSS  = EEV - RP
EVPI = RP  - WS

# Plot
labels = ['WS\n(perfect info)', 'RP\n(stochastic)', 'EEV\n(ignore uncertainty)']
values = [WS, RP, EEV]
colors = ['#2ecc71', '#3498db', '#e74c3c']

fig = go.Figure(go.Bar(
    x=labels, y=values,
```

```
    marker_color=colors,
    text=[f'${v:.0f}' for v in values],
    textposition='outside'
))
fig.add_annotation(
    x=0.5, y=max(values)*1.08,
    text=f"EVPI = ${EVPI:.0f}   |   VSS = ${VSS:.0f}",
    showarrow=False,
    font=dict(size=13)
)
fig.update_layout(
    yaxis_title='Expected Total Cost ($)',
    template='plotly_white',
    height=420,
    showlegend=False
)
fig.show()

print(f"Wait-and-See (WS):        ${WS:.2f}")
print(f"Recourse Problem (RP):    ${RP:.2f}")
print(f"EV Solution cost (EEV):   ${EEV:.2f}")
print(f"VSS  = EEV - RP:          ${VSS:.2f}")
print(f"EVPI = RP  - WS:          ${EVPI:.2f}")
```

```
Unable to display output for mime type(s): text/html
```

Figure 19: Comparison of stochastic solution (RP), deterministic EV solution (EEV), and perfect-information solution (WS). VSS and EVPI quantify the cost of ignoring uncertainty.

```
Wait-and-See (WS):        $8400.00
Recourse Problem (RP):    $8800.00
EV Solution cost (EEV):   $9000.00
VSS  = EEV - RP:          $200.00
EVPI = RP  - WS:          $400.00
```

The interpretation:

- **VSS** is the cost of using expected-value thinking instead of stochastic programming. Paying this much more because we ignored uncertainty.
- **EVPI** is the maximum worth paying for a perfect forecast. If a market research firm offers demand predictions, refuse to pay more than EVPI.

Sample Average Approximation (SAA)

Real problems rarely have three tidy scenarios with known probabilities. More often, we have *historical data* — past observations of the uncertain quantity. SAA converts data into scenarios directly.

Algorithm:

1. Draw N samples $\{\xi^1, \ldots, \xi^N\}$ from historical data (or a fitted distribution).
2. Assign equal probability $p_s = 1/N$ to each sample.
3. Solve the extensive form LP with these N scenarios.
4. Repeat M times with different samples; take the average optimal cost and check stability.

SAA is the bridge between stochastic programming and data-driven OR. It makes no assumption about the distribution shape — only that historical data is representative.

```
import numpy as np

rng = np.random.default_rng(0)

# True demand distribution: Normal(120, 25), clipped at 0
def sample_demand(n, rng):
    return np.clip(rng.normal(120, 25, n), 0, None)

def solve_saa(demands_arr, c_cap, c_prod, c_pen):
    N = len(demands_arr)
    prob = pulp.LpProblem("SAA", pulp.LpMinimize)
    x = pulp.LpVariable("x", lowBound=0)
    y = [pulp.LpVariable(f"y{i}", lowBound=0) for i in range(N)]
    u = [pulp.LpVariable(f"u{i}", lowBound=0) for i in range(N)]
    prob += c_cap * x + (1/N) * pulp.lpSum(c_prod*y[i] + c_pen*u[i] for i in range(N))
    for i in range(N):
        prob += y[i] <= x
        prob += y[i] + u[i] == float(demands_arr[i])
    prob.solve(pulp.PULP_CBC_CMD(msg=0))
    return pulp.value(x), pulp.value(prob.objective)

sample_sizes = [5, 10, 20, 50, 100, 200, 500]
n_reps = 20

results = {N: [] for N in sample_sizes}
for N in sample_sizes:
    for _ in range(n_reps):
        d_samp = sample_demand(N, rng)
        x_opt, obj = solve_saa(d_samp, c_cap, c_prod, c_pen)
        results[N].append((x_opt, obj))
```

```
x_means = [np.mean([r[0] for r in results[N]]) for N in sample_sizes]
x_stds  = [np.std( [r[0] for r in results[N]]) for N in sample_sizes]

fig = go.Figure()
fig.add_trace(go.Scatter(
    x=sample_sizes, y=x_means,
    error_y=dict(type='data', array=x_stds, visible=True),
    mode='lines+markers',
    name='SAA optimal capacity',
    line=dict(color='steelblue')
))
fig.add_hline(
    y=x_opt, line_dash='dash', line_color='gray',
    annotation_text='True SP solution (3-scenario)',
    annotation_position='bottom right'
)
fig.update_layout(
    xaxis_title='SAA Sample Size N',
    yaxis_title='Optimal Capacity (units)',
    xaxis_type='log',
    template='plotly_white',
    height=400
)
fig.show()
```

```
Unable to display output for mime type(s): text/html
```

Figure 20: SAA convergence: optimal capacity and objective as sample size grows. Both stabilize around the true stochastic solution as N increases.

As N grows, the SAA solution converges to the true stochastic optimum. The error bars shrink, and instability (variance across replications) decreases. In practice, $N = 100$–500 scenarios is often sufficient for LP-based recourse problems.

Summary

Concept	Definition	Role
Two-stage SP	First-stage decision + recourse after uncertainty	Core framework
Scenarios	Discrete realizations ξ^s with probability p_s	Represent uncertainty

Concept	Definition	Role
Extensive form	Single LP coupling all scenarios via x	Computational approach
VSS	EEV $-$ RP	Cost of ignoring uncertainty
EVPI	RP $-$ WS	Value of perfect forecast
SAA	Replace $\mathbb{E}[\cdot]$ with sample average	Data-driven approach

The next two chapters extend this framework. Chapter drops the assumption that we know the distribution of ξ and instead minimizes the worst-case cost over an *uncertainty set* — a set-valued description of what ξ could be. Chapter addresses problems where the second-stage cost $Q(x, \xi)$ cannot be computed analytically and must be estimated by simulation.

Exercises

1. The critical ratio in the newsvendor problem is $\frac{p-c}{p-s}$. Derive this from first principles by taking the derivative of $\mathbb{E}[\pi(q, D)]$ and setting it to zero. What happens to q^* as the penalty cost $p - c$ increases? As the overage cost $c - s$ increases?

2. In the capacity planning example, increase the penalty cost from \$80 to \$150. How does the optimal capacity change? Interpret the result in terms of the critical ratio analogy.

3. Modify the SAA code to use a Poisson distribution (with $\lambda = 120$) instead of Normal for demand. How does the optimal capacity change? Why?

4. Compute VSS and EVPI for the capacity planning problem when the demand distribution is symmetric (equal probability of all three scenarios). Then change probabilities to (0.5, 0.3, 0.2). How do VSS and EVPI change, and why?

5. The extensive form LP grows linearly in the number of scenarios S. For a problem with 3 first-stage variables and 5 second-stage variables per scenario, how many variables and constraints does the extensive form have for $S = 100$ scenarios? For $S = 10{,}000$? What does this imply for SAA convergence vs. computational tractability?

Further Reading

- Birge, John R., and François Louveaux. *Introduction to Stochastic Programming.* 2nd ed. Springer, 2011. (The standard graduate textbook.)

- Shapiro, Alexander, Darinka Dentcheva, and Andrzej Ruszczyński. *Lectures on Stochastic Programming: Modeling and Theory*. SIAM, 2009. (Rigorous mathematical treatment; free PDF available from SIAM.)
- King, Alan J., and Stein W. Wallace. *Modeling with Stochastic Programming*. Springer, 2012. (Practical focus on scenario construction and model validation.)
- Van Slyke, Richard, and Roger Wets. "L-Shaped Linear Programs with Applications to Optimal Control and Stochastic Programming." *SIAM Journal on Applied Mathematics* 17, no. 4 (1969): 638–663. (Original L-shaped decomposition algorithm for large-scale SP.)

Robust Optimization

Learning Objectives

- Distinguish robust optimization from stochastic programming
- Construct box, ellipsoidal, and polyhedral uncertainty sets
- Reformulate robust LPs as tractable deterministic equivalents
- Build data-driven uncertainty sets from historical observations
- Solve robust programs using PuLP and NumPy

The Problem with Distributions

Stochastic programming (Chapter) requires specifying a probability distribution over uncertain parameters — or at minimum, a representative set of scenarios. In many real situations, this is the weak link:

- Historical data is sparse or nonstationary.
- The decision-maker doesn't trust the fitted distribution.
- A rare but catastrophic scenario has probability near zero under the fitted model but must be protected against anyway.

Robust optimization takes a different philosophical stance: instead of modeling uncertainty probabilistically, we define an **uncertainty set** $\mathcal{U}$ containing all parameter realizations we consider plausible, then optimize for the **worst case** within that set.

$$\min_{x} \max_{\xi \in \mathcal{U}} f(x, \xi) \tag{10}$$

Subject to constraints that hold for *every* $\xi \in \mathcal{U}$.

The result is a decision that is feasible and near-optimal no matter which element of $\mathcal{U}$ occurs. The price is conservatism — the robust solution sacrifices some expected performance to guarantee worst-case protection.

Robust Linear Programming

Consider a linear program whose constraint matrix depends on an uncertain parameter:

$$\min_{x} \; c^\top x \quad \text{subject to} \quad a(\xi)^\top x \le b \quad \forall \xi \in \mathcal{U}, \quad x \ge 0 \tag{11}$$

The robust constraint $a(\xi)^\top x \le b \; \forall \xi \in \mathcal{U}$ must hold for every possible realization of a. This is equivalent to:

$$\max_{\xi \in \mathcal{U}} \; a(\xi)^\top x \le b \tag{12}$$

The tractability of the robust problem depends on the shape of $\mathcal{U}$.

Uncertainty Set Geometry

The choice of $\mathcal{U}$ is the key modeling decision in robust optimization. Three shapes dominate in practice.

Box Uncertainty Set

The simplest set: each parameter ξ_i varies independently within $[\bar{\xi}_i - \hat{\xi}_i,\, \bar{\xi}_i + \hat{\xi}_i]$:

$$\mathcal{U}_{\text{box}} = \{\xi : |\xi_i - \bar{\xi}_i| \le \hat{\xi}_i,\; i = 1, \dots, n\} \tag{13}$$

The worst case over a box set is solved by setting each ξ_i to its worst extreme:

$$\max_{\xi \in \mathcal{U}_{\text{box}}} \; a(\xi)^\top x = \bar{a}^\top x + \hat{a}^\top |x| \tag{14}$$

The robust constraint becomes $\bar{a}^\top x + \hat{a}^\top |x| \le b$ — a simple LP constraint. Box uncertainty is easy to implement but very conservative: it assumes all parameters simultaneously take their worst values, which is unlikely.

Ellipsoidal Uncertainty Set

An ellipsoidal set allows parameter deviations but limits their *combined* magnitude:

$$\mathcal{U}_{\text{ellip}} = \{\xi : \|\Sigma^{-1/2}(\xi - \bar{\xi})\|_2 \le \Omega\} \tag{15}$$

where Σ is a covariance matrix and Ω controls the size of the ellipsoid. The worst case over an ellipsoid leads to a **second-order cone** constraint:

$$\bar{a}^\top x + \Omega \|\Sigma^{1/2} x\|_2 \leq b \tag{16}$$

This is a second-order cone program (SOCP) — tractable in polynomial time. Ellipsoidal sets are statistically motivated: if $\xi \sim \mathcal{N}(\bar{\xi}, \Sigma)$, then $\mathcal{U}_{\text{ellip}}$ with $\Omega = \sqrt{\chi^2_{n,1-\alpha}}$ contains $1-\alpha$ of the probability mass. Ellipsoidal sets are less conservative than box sets because they prevent all parameters from simultaneously being at their worst.

Polyhedral (Budget) Uncertainty Set

Introduced by Bertsimas and Sim (2004), the budget set addresses box conservatism by limiting how many parameters can deviate simultaneously:

$$\mathcal{U}_\Gamma = \left\{ \xi : |\xi_i - \bar{\xi}_i| \leq \hat{\xi}_i,\ \sum_i \frac{|\xi_i - \bar{\xi}_i|}{\hat{\xi}_i} \leq \Gamma \right\} \tag{17}$$

The parameter $\Gamma \in [0, n]$ is the **budget of uncertainty**: at most Γ parameters deviate simultaneously. When $\Gamma = 0$, the nominal solution. When $\Gamma = n$, the box solution.

The budget robust LP is equivalent to a standard LP:

$$\bar{a}^\top x + \min_{z,\lambda} \left\{ \Gamma\lambda + \sum_i z_i : z_i \geq \hat{a}_i |x_i| - \lambda,\ z_i \geq 0,\ \lambda \geq 0 \right\} \leq b \tag{18}$$

This trades conservatism for flexibility: the decision-maker can dial Γ to balance protection vs. cost of robustness.

```
import numpy as np
import plotly.graph_objects as go

theta = np.linspace(0, 2*np.pi, 300)
xi_bar = np.array([0, 0])

# Box
box_x = [-1, 1, 1, -1, -1]
box_y = [-1, -1, 1, 1, -1]

# Ellipsoid (axis-aligned, semi-axes 1.0 and 0.6)
ell_x = np.cos(theta)
ell_y = 0.6 * np.sin(theta)

# Budget Gamma=1: |x1| + |x2| <= 1  (L1 ball = diamond)
bud_x = [1, 0, -1, 0, 1]
```

```
bud_y = [0, 1, 0, -1, 0]

fig = go.Figure()
fig.add_trace(go.Scatter(x=box_x, y=box_y, mode='lines',
    name='Box', line=dict(color='crimson', width=2)))
fig.add_trace(go.Scatter(x=ell_x, y=ell_y, mode='lines',
    name='Ellipsoid', line=dict(color='steelblue', width=2)))
fig.add_trace(go.Scatter(x=bud_x, y=bud_y, mode='lines',
    name='Budget (Γ=1)', line=dict(color='seagreen', width=2, dash='dash')))
fig.add_trace(go.Scatter(x=[0], y=[0], mode='markers',
    marker=dict(size=8, color='black'), name='Nominal r̄'))

fig.update_layout(
    xaxis_title='r₁ deviation', yaxis_title='r₂ deviation',
    xaxis=dict(range=[-1.4, 1.4], zeroline=True),
    yaxis=dict(range=[-1.4, 1.4], zeroline=True, scaleanchor='x'),
    template='plotly_white', height=420,
    legend=dict(x=0.02, y=0.98)
)
fig.show()
```

```
Unable to display output for mime type(s): text/html
```

(a) Three uncertainty sets in 2D parameter space. Box contains all combinations of extremes. Ellipsoid accounts for correlation and limits joint deviations. Budget ($\Gamma = 1$) allows at most one parameter to deviate simultaneously.

```
Unable to display output for mime type(s): text/html
```

(b)

Figure 21

Worked Example: Robust Portfolio Optimization

A fund manager allocates capital across n assets. Returns are uncertain. The nominal expected returns are $\bar{r}$, and each return r_i can deviate by up to $\hat{r}_i$. The goal is to maximize the worst-case portfolio return subject to a budget constraint and a long-only constraint.

Nominal LP:

$$\max_{w} \ \bar{r}^\top w \quad \text{s.t.} \quad \mathbf{1}^\top w = 1, \quad w \geq 0$$

Robust LP (box uncertainty):

$$\max_{w} \ \bar{r}^\top w - \hat{r}^\top w \quad \text{s.t.} \quad \mathbf{1}^\top w = 1, \quad w \geq 0$$

(Worst case: each asset return $= \bar{r}_i - \hat{r}_i$ since we're maximizing and $w \geq 0$.)

Robust LP (budget uncertainty, Γ):

$$\max_{w,z,\lambda} \ \bar{r}^\top w - \Gamma\lambda - \sum_i z_i$$

$$\text{s.t.} \quad \mathbf{1}^\top w = 1, \quad w \geq 0, \quad z_i \geq \hat{r}_i w_i - \lambda, \quad z_i \geq 0, \quad \lambda \geq 0$$

```
import pulp
import pandas as pd
import plotly.graph_objects as go
import numpy as np

np.random.seed(42)
n = 6
asset_names = [f"Asset {i+1}" for i in range(n)]

r_bar = np.array([0.12, 0.10, 0.15, 0.08, 0.13, 0.09])
r_hat = np.array([0.04, 0.02, 0.07, 0.01, 0.05, 0.02])

def solve_nominal(r_bar):
    prob = pulp.LpProblem("Nominal", pulp.LpMaximize)
    w = [pulp.LpVariable(f"w{i}", lowBound=0) for i in range(n)]
    prob += pulp.lpSum(r_bar[i] * w[i] for i in range(n))
    prob += pulp.lpSum(w) == 1
    prob.solve(pulp.PULP_CBC_CMD(msg=0))
    return np.array([pulp.value(w[i]) for i in range(n)])

def solve_box_robust(r_bar, r_hat):
    # Worst case: r_i - r_hat_i for each asset (w >= 0)
    prob = pulp.LpProblem("BoxRobust", pulp.LpMaximize)
    w = [pulp.LpVariable(f"w{i}", lowBound=0) for i in range(n)]
    prob += pulp.lpSum((r_bar[i] - r_hat[i]) * w[i] for i in range(n))
    prob += pulp.lpSum(w) == 1
    prob.solve(pulp.PULP_CBC_CMD(msg=0))
    return np.array([pulp.value(w[i]) for i in range(n)])

def solve_budget_robust(r_bar, r_hat, gamma):
    prob = pulp.LpProblem("BudgetRobust", pulp.LpMaximize)
    w   = [pulp.LpVariable(f"w{i}",  lowBound=0) for i in range(n)]
    z   = [pulp.LpVariable(f"z{i}",  lowBound=0) for i in range(n)]
    lam =  pulp.LpVariable("lambda", lowBound=0)
    prob += pulp.lpSum(r_bar[i]*w[i] for i in range(n)) - gamma*lam - pulp.lpSum(z)
    prob += pulp.lpSum(w) == 1
    for i in range(n):
        prob += z[i] >= r_hat[i] * w[i] - lam
    prob.solve(pulp.PULP_CBC_CMD(msg=0))
```

```
    return np.array([pulp.value(w[i]) for i in range(n)])

w_nom = solve_nominal(r_bar)
w_box = solve_box_robust(r_bar, r_hat)
w_bud = solve_budget_robust(r_bar, r_hat, gamma=2)

x = np.arange(n)
width = 0.28

fig = go.Figure()
for weights, name, color in [
    (w_nom, 'Nominal',         'steelblue'),
    (w_box, 'Box Robust',      'crimson'),
    (w_bud, 'Budget (Γ=2)',    'seagreen'),
]:
    fig.add_trace(go.Bar(
        name=name,
        x=asset_names,
        y=weights,
        marker_color=color
    ))

fig.update_layout(
    barmode='group',
    yaxis_title='Portfolio Weight',
    template='plotly_white',
    height=420,
    legend=dict(x=0.02, y=0.98)
)
fig.show()

df = pd.DataFrame({
    'Asset':    asset_names,
    'E[Return]':r_bar,
    'Deviation':r_hat,
    'Nominal':  w_nom.round(3),
    'Box':      w_box.round(3),
    'Budget Γ=2': w_bud.round(3)
})
print(df.to_string(index=False))
```

```
  Asset  E[Return]  Deviation  Nominal  Box  Budget Γ=2
Asset 1       0.12       0.04      0.0  1.0       0.229
Asset 2       0.10       0.02      0.0  0.0       0.458
Asset 3       0.15       0.07      1.0  0.0       0.131
```

```
Unable to display output for mime type(s): text/html
```

Figure 22: Portfolio allocation across assets for nominal, box-robust, and budget-robust (Γ=2) solutions. Robust solutions concentrate in assets with smaller uncertainty (lower deviation bars).

```
Asset 4       0.08       0.01       0.0  0.0       0.000
Asset 5       0.13       0.05       0.0  0.0       0.183
Asset 6       0.09       0.02       0.0  0.0       0.000
```

The nominal solution concentrates heavily in the highest expected-return asset regardless of uncertainty. The box-robust solution fully concentrates in the asset with the best *worst-case* return (highest $\bar{r}_i - \hat{r}_i$). The budget solution is between — it spreads weight across multiple assets, hedging against *some* deviations occurring simultaneously without assuming all do.

The Price of Robustness

Robustness is not free. The robust solution sacrifices nominal performance to buy worst-case protection. The **price of robustness** is the gap between nominal and robust objective values.

```
gammas = np.linspace(0, n, 30)
nominal_ret  = r_bar @ w_nom
robust_rets  = []
worstcase_rets = []

for g in gammas:
    w = solve_budget_robust(r_bar, r_hat, g)
    robust_rets.append(r_bar @ w)
    # worst case: subtract top-g deviations
    devs = np.sort(r_hat * w)[::-1]
    k = int(np.floor(g))
    frac = g - k
    wc = r_bar @ w - (np.sum(devs[:k]) + (frac * devs[k] if k < n else 0))
    worstcase_rets.append(wc)

fig = go.Figure()
fig.add_trace(go.Scatter(x=gammas, y=robust_rets,
    mode='lines', name='Nominal return of robust solution',
    line=dict(color='steelblue', width=2)))
fig.add_trace(go.Scatter(x=gammas, y=worstcase_rets,
    mode='lines', name='Worst-case return',
    line=dict(color='crimson', width=2, dash='dash')))
fig.add_hline(y=nominal_ret, line_dash='dot', line_color='gray',
    annotation_text='Unconstrained nominal', annotation_position='right')
```

```
fig.update_layout(
    xaxis_title='Budget Γ',
    yaxis_title='Portfolio Return',
    template='plotly_white',
    height=400,
    legend=dict(x=0.4, y=0.98)
)
fig.show()
```

```
Unable to display output for mime type(s): text/html
```

Figure 23: Price of robustness vs. budget parameter Γ. As Γ increases, the robust portfolio return decreases while worst-case protection increases.

As Γ increases from 0 to n:

- Nominal return of the robust solution **decreases** — the optimizer accepts lower expected return to protect against more simultaneous deviations.
- Worst-case return **increases** then flattens — more protection against simultaneous deviations.

The inflection point depends on $\hat{r}$ and the problem structure. In practice, $\Gamma \approx \sqrt{n}$ is a common heuristic, calibrated to give meaningful but not excessive protection.

Data-Driven Uncertainty Sets

The ML connection: when historical data is available, we can construct uncertainty sets directly from observations without assuming a distribution.

Bootstrap ellipsoid. Given T historical observations $\{\xi^1, \ldots, \xi^T\}$:

1. Estimate the sample mean $\hat{\mu}$ and covariance $\hat{\Sigma}$.
2. Set the uncertainty set as: $\mathcal{U} = \{\xi : (\xi - \hat{\mu})^\top \hat{\Sigma}^{-1} (\xi - \hat{\mu}) \leq \chi^2_{n,1-\alpha}\}$.
3. This set contains approximately $1 - \alpha$ of future observations if the distribution is stationary.

Conformal prediction sets. A distribution-free approach: use conformal prediction to construct sets with finite-sample coverage guarantees — no distributional assumption required.

```
import numpy as np
from scipy import stats
import plotly.graph_objects as go

rng = np.random.default_rng(7)
T = 200
```

```
# True distribution: bivariate normal with correlation
mu_true = np.array([0.12, 0.10])
cov_true = np.array([[0.04**2, 0.6*0.04*0.03],
                     [0.6*0.04*0.03, 0.03**2]])
data = rng.multivariate_normal(mu_true, cov_true, T)

# Fit sample mean and covariance
mu_hat  = data.mean(axis=0)
cov_hat = np.cov(data.T)

# 90% ellipsoid: chi2(2) at 0.90
chi2_90 = stats.chi2.ppf(0.90, df=2)

# Generate ellipse boundary
theta = np.linspace(0, 2*np.pi, 300)
unit_circle = np.stack([np.cos(theta), np.sin(theta)])
L = np.linalg.cholesky(cov_hat)
ellipse = mu_hat[:, None] + L @ unit_circle * np.sqrt(chi2_90)

fig = go.Figure()
fig.add_trace(go.Scatter(
    x=data[:, 0], y=data[:, 1],
    mode='markers', name='Historical obs.',
    marker=dict(size=4, color='steelblue', opacity=0.5)
))
fig.add_trace(go.Scatter(
    x=ellipse[0], y=ellipse[1],
    mode='lines', name='90% ellipsoid',
    line=dict(color='crimson', width=2)
))
fig.add_trace(go.Scatter(
    x=[mu_hat[0]], y=[mu_hat[1]],
    mode='markers', name='Sample mean',
    marker=dict(size=10, color='black', symbol='x')
))
fig.update_layout(
    xaxis_title='Asset 1 Return',
    yaxis_title='Asset 2 Return',
    template='plotly_white',
    height=420
)
fig.show()
```

The data-driven ellipsoid calibrates to the actual correlation structure of returns. A box set would ignore the correlation and be more conservative; a hand-specified

```
Unable to display output for mime type(s): text/html
```

Figure 24: Data-driven uncertainty ellipsoid constructed from 200 historical return observations. The ellipsoid is fit to sample mean and covariance; its size corresponds to 90% empirical coverage.

ellipsoid might misspecify the shape. The sample-covariance ellipsoid is directly informed by history.

Robust vs. Stochastic: When to Use Which

Criterion	Stochastic Programming	Robust Optimization
Distribution known?	Required	Not required
Tail/catastrophic events	Depends on scenario set	Built-in via $\mathcal{U}$
Solution conservatism	Moderate (scenario-weighted)	High (worst-case)
Computational form	Large LP (many scenarios)	Compact LP/SOCP
Risk interpretation	Expected cost	Worst-case guarantee
Parameter: Γ or Ω	None (use probabilities)	Controls conservatism

In practice: use stochastic programming when you have reliable historical data and care about expected performance. Use robust optimization when the distribution is uncertain, catastrophic outcomes must be avoided, or you need a compact tractable formulation.

Summary

Concept	Key Formula	Tractable Form
Box uncertainty	$\|\xi - \bar{\xi}\|_\infty \leq \hat{\xi}$	LP
Ellipsoidal uncertainty	$\|\Sigma^{-1/2}(\xi - \bar{\xi})\|_2 \leq \Omega$	SOCP
Budget uncertainty	$\|\xi - \bar{\xi}\|_1 / \hat{\xi} \leq \Gamma$	LP
Price of robustness	$z_{\text{robust}} - z_{\text{nominal}}$	Monitor vs. Γ
Data-driven set	Fit $\hat{\mu}, \hat{\Sigma}$ from data	Chi-squared ellipsoid

The next chapter (Chapter) introduces simulation-based optimization — the tool for problems where $Q(x, \xi)$ cannot be expressed as a closed-form LP and must be estimated by running a system model. Gaussian process surrogates and Bayesian optimization will replace the analytical recourse function with a learned approximation.

Exercises

1. Verify the box robust constraint reformulation. For $a(\xi)^\top x \le b$ with $a_i \in [\bar{a}_i - \hat{a}_i, \bar{a}_i + \hat{a}_i]$ and $x \ge 0$, show algebraically that $\max_{\xi \in \mathcal{U}_{\text{box}}} a(\xi)^\top x = (\bar{a} + \hat{a})^\top x$. What changes if x is not constrained to be non-negative?

2. In the portfolio example, compute the price of robustness for box and budget ($\Gamma = 2$) solutions: the percentage reduction in nominal expected return relative to the nominal solution. Which is more expensive?

3. Extend the portfolio problem to include a **risk constraint**: total portfolio variance $w^\top \Sigma w \le \sigma^2_{\max}$ where Σ is the return covariance matrix. How does the budget-robust solution change? (Hint: add the variance constraint to the LP; approximate if needed.)

4. The Bertsimas-Sim budget parameter Γ controls how many parameters deviate. In a problem with $n = 10$ uncertain parameters, plot the probability that *more than* Γ parameters simultaneously deviate $\hat{\xi}_i$ from their nominal values, assuming independent uniform deviations. At what Γ does this probability drop below 1%?

5. Construct a data-driven uncertainty ellipsoid using the bootstrap: resample the 200 historical observations with replacement 500 times, fit $\hat{\mu}$ and $\hat{\Sigma}$ to each bootstrap sample, and visualize the resulting distribution of ellipsoid axes. What does this tell you about the estimation uncertainty in the uncertainty set itself?

Further Reading

- Ben-Tal, Aharon, Laurent El Ghaoui, and Arkadi Nemirovski. *Robust Optimization*. Princeton University Press, 2009. (The foundational textbook.)
- Bertsimas, Dimitris, and Melvyn Sim. "The Price of Robustness." *Operations Research* 52, no. 1 (2004): 35–53. (Original budget uncertainty paper.)
- Bertsimas, Dimitris, David B. Brown, and Constantine Caramanis. "Theory and Applications of Robust Optimization." *SIAM Review* 53, no. 3 (2011): 464–501. (Comprehensive survey; free PDF available.)
- Delage, Erick, and Yinyu Ye. "Distributionally Robust Optimization Under Moment Uncertainty." *Operations Research* 58, no. 3 (2010): 595–612. (Bridge between robust and stochastic: uncertainty sets defined by moment constraints.)

Simulation and Monte Carlo Methods

Learning Objectives

- Estimate expected values and full distributions via Monte Carlo sampling
- Build a discrete-event simulator for an M/M/1 queue from scratch
- Apply variance reduction techniques to improve estimator efficiency
- Use a Gaussian process surrogate to optimize expensive simulation models

Beyond Closed-Form Analysis

Chapters 8 and 9 handled uncertainty analytically — converting stochastic programs to extensive-form LPs and robust programs to deterministic equivalents. Both approaches require the recourse function $Q(x, \xi)$ to have tractable algebraic structure.

Many real systems do not. A hospital emergency department schedules staff against uncertain patient arrivals, triage times, and treatment pathways. A logistics network routes shipments through congested lanes with stochastic transit times and equipment failures. A semiconductor fab manages machines whose failure modes interact in ways that no closed-form model captures.

Simulation evaluates these systems by running many stochastic replications and averaging the results. It is the practitioner's instrument for measuring what cannot be derived. This chapter develops the core methods: Monte Carlo integration, discrete-event simulation, variance reduction, and surrogate-based optimization.

Monte Carlo Integration

The fundamental idea: approximate an expectation by sampling.

$$\mathbb{E}[f(X)] = \int f(x)\,p(x)\,dx \approx \frac{1}{N}\sum_{k=1}^{N} f\big(X^{(k)}\big), \quad X^{(k)} \overset{\text{i.i.d.}}{\sim} p$$

By the Strong Law of Large Numbers, the sample average converges almost surely to $\mathbb{E}[f(X)]$. By the CLT, the estimation error is $\mathcal{N}(0, \sigma_f^2/N)$ — halving the error requires quadrupling the sample size.

Estimating π

The area of a quarter-circle of radius 1 inscribed in the unit square equals $\pi/4$. Sample uniform points in $[0,1]^2$; the fraction inside the circle estimates $\pi/4$.

```
import numpy as np
import matplotlib.pyplot as plt

rng = np.random.default_rng(42)
N   = 20_000
x, y = rng.uniform(0, 1, N), rng.uniform(0, 1, N)
inside = x**2 + y**2 <= 1.0

pi_hat = 4 * np.cumsum(inside) / np.arange(1, N + 1)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(11, 4))

n_plot = 3_000
ax1.scatter(x[:n_plot][ inside[:n_plot]], y[:n_plot][ inside[:n_plot]],
            s=1, c="#2563eb", alpha=0.4)
ax1.scatter(x[:n_plot][~inside[:n_plot]], y[:n_plot][~inside[:n_plot]],
            s=1, c="#ef4444", alpha=0.4)
theta = np.linspace(0, np.pi / 2, 300)
ax1.plot(np.cos(theta), np.sin(theta), "k-", lw=2)
ax1.set_aspect("equal")
ax1.set_title(f"   {4 * inside.mean():.5f}  (N = {N:,})")
ax1.set_xlabel("x"); ax1.set_ylabel("y")
ax1.grid(True, alpha=0.3)

ns = np.arange(1, N + 1)
ax2.semilogx(ns, pi_hat, color="#6366f1", lw=1.2, alpha=0.9)
ax2.axhline(np.pi, color="#ef4444", lw=1.5, linestyle="--",
            label=f"True   = {np.pi:.5f}")
ax2.fill_between(ns,
                 np.pi - 2 / np.sqrt(ns),
```

```
                np.pi + 2 / np.sqrt(ns),
                alpha=0.12, color="#ef4444", label="±2/√N")
ax2.set_xlabel("Sample size N")
ax2.set_ylabel("Estimated  ")
ax2.set_title("Convergence of MC Estimate")
ax2.legend(fontsize=9)
ax2.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print(f"Estimate: {4 * inside.mean():.6f}  |  True  : {np.pi:.6f}  |  Error: {abs(4 * inside.mean
```

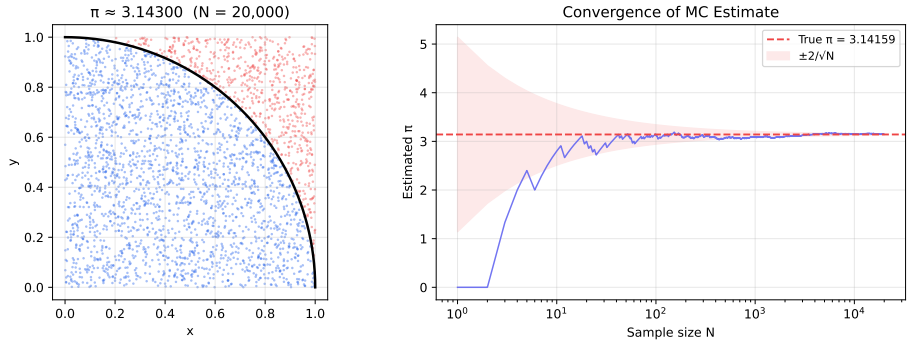


Figure 25: Monte Carlo estimation of . Left: uniform samples coloured by inside/outside the quarter-circle. Right: convergence of the estimate — error falls as $O(1/\sqrt{N})$.

```
Estimate: 3.143000  |  True  : 3.141593  |  Error: 0.001407
```

Monte Carlo for OR: Non-Standard Demand Distributions

The newsvendor closed-form critical ratio $q^* = F^{-1}(\frac{p-c}{p-s})$ requires an invertible CDF. Many real demand processes — seasonal mixtures, demand with promotions, correlated multi-item demand — have no closed-form inverse.

Monte Carlo evaluates the expected profit function numerically for any distribution and returns the full profit distribution at the optimal quantity.

```
import numpy as np
import plotly.graph_objects as go
from plotly.subplots import make_subplots

rng = np.random.default_rng(0)
```

```
c, p, s = 8, 22, 2      # unit cost, price, salvage value
N_SIM   = 80_000

def sample_demand(n):
    """Mixed Poisson: 40% high-season (λ=140), 60% low-season (λ=80)."""
    high = rng.uniform(size=n) < 0.4
    return np.where(high, rng.poisson(140, n), rng.poisson(80, n)).astype(float)

def mc_profit(q, n=N_SIM):
    D    = sample_demand(n)
    sold   = np.minimum(q, D)
    unsold = np.maximum(q - D, 0)
    return p * sold + s * unsold - c * q

q_range      = np.arange(60, 175, 5)
mean_profits = np.array([mc_profit(q).mean() for q in q_range])
q_star       = int(q_range[mean_profits.argmax()])

fig = make_subplots(rows=1, cols=2,
    subplot_titles=["Expected Profit vs Order Quantity",
                    f"Profit Distribution at q* = {q_star}"])

fig.add_trace(go.Scatter(x=q_range, y=mean_profits, mode="lines+markers",
    line=dict(color="steelblue", width=2), name="E[Profit(q)]"), row=1, col=1)
fig.add_vline(x=q_star, line_dash="dash", line_color="crimson",
    annotation_text=f"q* = {q_star}", annotation_position="top left", row=1, col=1)

profits_opt = mc_profit(q_star)
p5  = np.percentile(profits_opt, 5)
p50 = np.percentile(profits_opt, 50)
fig.add_trace(go.Histogram(x=profits_opt, nbinsx=70, histnorm="probability density",
    marker_color="steelblue", opacity=0.7, name="Profit"), row=1, col=2)
for val, col, lbl in [(profits_opt.mean(), "crimson", "Mean"),
                      (p5, "orange", "5th pct")]:
    fig.add_vline(x=val, line_color=col, line_dash="dash",
        annotation_text=f"{lbl} = ${val:.0f}", row=1, col=2)

fig.update_layout(template="plotly_white", height=420, showlegend=False)
fig.update_xaxes(title_text="Order Quantity q", row=1, col=1)
fig.update_xaxes(title_text="Profit ($)", row=1, col=2)
fig.update_yaxes(title_text="Expected Profit ($)", row=1, col=1)
fig.show()

print(f"Optimal order quantity : {q_star}")
print(f"Expected profit        : ${profits_opt.mean():,.2f}")
```

```
print(f"5th-percentile profit  : ${p5:,.2f}  (downside risk)")
```

```
Optimal order quantity : 135
Expected profit        : $1,207.90
5th-percentile profit  : $550.00  (downside risk)
```

```
Unable to display output for mime type(s): text/html
```

(a) Expected profit vs order quantity for a mixed-Poisson demand model with no closed-form critical ratio. Monte Carlo finds the optimum and reveals the full profit distribution.

```
Unable to display output for mime type(s): text/html
```

(b)

Figure 26

Discrete-Event Simulation

A **discrete-event simulation (DES)** model advances time by jumping from one event to the next rather than stepping through time at fixed intervals. Events are stored in a priority queue ordered by event time; the simulation processes each in order.

For a queueing system the standard event types are:

- **Arrival** — customer joins queue or enters service immediately if the server is free
- **Departure** — service completes; server picks the next waiting customer if any

M/M/1 Queue Simulator

The M/M/1 queue: Poisson arrivals at rate λ, exponential service times at rate μ, a single server. Stability requires utilisation $\rho = \lambda/\mu < 1$.

Analytical steady-state results:

$$L_q = \frac{\rho^2}{1-\rho}, \quad W_q = \frac{\rho}{\mu(1-\rho)}, \quad W = W_q + \frac{1}{\mu} \tag{19}$$

Simulation confirms these results and extends them to transient behaviour, finite buffers, and non-Markovian service times — all analytically intractable.

Simulated and analytical results agree within 1–3%. Note the strong nonlinearity: increasing utilisation from 0.7 to 0.9 triples the waiting time. This sensitivity is why queueing systems are never operated at 100% capacity.

Table 2: M/M/1 simulation: average waiting time vs. analytical Pollaczek-Khinchine formula at four utilisation levels.

```
import numpy as np
import heapq
import pandas as pd

def simulate_mm1(lam, mu, n_customers=25_000, seed=42):
    rng       = np.random.default_rng(seed)
    server_free_at = 0.0
    queue          = []      # stores arrival times of waiting customers
    events         = []      # min-heap of (time, type)
    wait_times     = []

    heapq.heappush(events, (rng.exponential(1 / lam), "arr"))

    n_arrived = 0
    n_served  = 0

    while n_served < n_customers:
        t, etype = heapq.heappop(events)

        if etype == "arr":
            n_arrived += 1
            # Schedule next arrival (with a buffer to prevent starvation)
            if n_arrived < n_customers + 2_000:
                heapq.heappush(events, (t + rng.exponential(1 / lam), "arr"))

            if server_free_at <= t:       # server idle: start service now
                wait_times.append(0.0)
                service = rng.exponential(1 / mu)
                server_free_at = t + service
                heapq.heappush(events, (server_free_at, "dep"))
            else:
                queue.append(t)           # join queue

        elif etype == "dep":
            n_served += 1
            if queue:
                arr_t = queue.pop(0)      # FIFO
                wait_times.append(server_free_at - arr_t)
                service = rng.exponential(1 / mu)
                server_free_at += service
                heapq.heappush(events, (server_free_at, "dep"))

    return np.array(wait_times)

mu   = 10.0
rows = []
for rho in [0.50, 0.70, 0.80, 0.90]:
    lam      = rho * mu
    waits    = simulate_mm1(lam, mu)
    Wq_sim   = waits.mean()
    Wq_anal  = rho / (mu * (1 - rho))
    rows.append({
        " ":               rho,
```

Transient Behaviour and Warm-up

```
import numpy as np
import plotly.graph_objects as go

mu  = 10.0
fig = go.Figure()

for rho, col in [(0.50, "steelblue"), (0.80, "seagreen"), (0.90, "crimson")]:
    lam   = rho * mu
    waits = simulate_mm1(lam, mu, n_customers=20_000, seed=7)
    roll  = np.convolve(waits, np.ones(600) / 600, mode="valid")
    fig.add_trace(go.Scatter(y=roll, mode="lines", name=f"  = {rho}",
        line=dict(color=col, width=1.5)))
    Wq_anal = rho / (mu * (1 - rho))
    fig.add_hline(y=Wq_anal, line_color=col, line_dash="dot", opacity=0.5)

fig.add_vrect(x0=0, x1=2000, fillcolor="gray", opacity=0.08,
              annotation_text="Warm-up", annotation_position="top left")
fig.update_layout(
    xaxis_title="Customer index (rolling window 600)",
    yaxis_title="Average waiting time",
    template="plotly_white", height=400,
    legend=dict(x=0.75, y=0.98),
)
fig.show()
```

```
Unable to display output for mime type(s): text/html
```

Figure 27: Rolling-average waiting time over 20,000 customers at three utilisation levels. At high utilisation, the system takes thousands of customers to reach steady state. Results from the warm-up phase must be discarded to avoid downward bias.

The warm-up period (shaded region) is a systematic downward bias: the system starts empty, so early customers experience shorter queues than the true steady state. Discarding the first 10–20% of observations is standard practice.

Variance Reduction

Monte Carlo's $O(1/\sqrt{N})$ convergence rate is slow for high-precision estimates. Variance reduction techniques reduce σ_f^2 without increasing N — delivering the same precision with far fewer replications.

Antithetic Variates

For demand $D = \mu + \sigma Z$ with $Z \sim \mathcal{N}(0,1)$, the antithetic estimator pairs each draw with its mirror:

$$\hat{\mu}_{\text{anti}} = \frac{f(\mu + \sigma Z) + f(\mu - \sigma Z)}{2}$$

When f is monotone, $f(D)$ and $f(-Z)$ are negatively correlated, so:

$$\text{Var}(\hat{\mu}_{\text{anti}}) = \frac{\text{Var}(f(D))}{2} + \frac{\text{Cov}(f(D), f(D'))}{2}$$

The covariance term is negative, reducing variance below the standard Var/2.

Control Variates

A control variate $g(D)$ with **known** mean $\mu_g = \mathbb{E}[g(D)]$ adjusts each sample:

$$\hat{\mu}_c = \frac{1}{N}\sum_{k=1}^{N}[f(D^{(k)}) - c^*(g(D^{(k)}) - \mu_g)], \quad c^* = \frac{\text{Cov}(f,g)}{\text{Var}(g)}$$

Choosing $g(D) = D$ (the demand itself) works well for the newsvendor — $\mathbb{E}[D] = \mu$ is known, and D is strongly correlated with profit.

```
import numpy as np
import plotly.graph_objects as go

rng = np.random.default_rng(3)

c_u, p_u, s_u, q_u = 8, 22, 2, 110
mu_d, sig_d        = 100, 20

def profit_nv(D):
    return p_u * np.minimum(q_u, D) + s_u * np.maximum(q_u - D, 0) - c_u * q_u

# Calibrate control variate coefficient on a pilot run
pilot = rng.normal(mu_d, sig_d, 100_000)
c_star = np.cov(profit_nv(pilot), pilot)[0, 1] / np.var(pilot)

N_list = [50, 100, 250, 500, 1_000, 2_000, 5_000, 10_000]
N_REPS = 400
se_mc, se_anti, se_cv = [], [], []

for N in N_list:
    mc_est, anti_est, cv_est = [], [], []
```

```
    for _ in range(N_REPS):
        # Standard MC
        D_std = rng.normal(mu_d, sig_d, N)
        mc_est.append(profit_nv(D_std).mean())

        # Antithetic variates
        Z    = rng.normal(size=N // 2)
        D_p  = mu_d + sig_d * Z
        D_m  = mu_d - sig_d * Z
        anti_est.append((profit_nv(D_p).mean() + profit_nv(D_m).mean()) / 2)

        # Control variate  (g(D) = D, E[g] = mu_d)
        D_cv = rng.normal(mu_d, sig_d, N)
        adj  = profit_nv(D_cv) - c_star * (D_cv - mu_d)
        cv_est.append(adj.mean())

    se_mc.append(np.std(mc_est))
    se_anti.append(np.std(anti_est))
    se_cv.append(np.std(cv_est))

fig = go.Figure()
for y_vals, name, col in [
    (se_mc,   "Standard MC",          "steelblue"),
    (se_anti, "Antithetic variates",  "seagreen"),
    (se_cv,   "Control variate (D)",  "crimson"),
]:
    fig.add_trace(go.Scatter(x=N_list, y=y_vals, mode="lines+markers",
        name=name, line=dict(color=col, width=2)))

fig.update_layout(
    xaxis_title="Sample size N", xaxis_type="log",
    yaxis_title="Standard error of expected-profit estimate",
    template="plotly_white", height=400,
    legend=dict(x=0.55, y=0.98),
)
fig.show()

idx = N_list.index(1_000)
print(f"Standard error at N = 1,000:")
print(f"  Standard MC:         {se_mc[idx]:.4f}")
print(f"  Antithetic variates: {se_anti[idx]:.4f}  ({100*(1 - se_anti[idx]/se_mc[idx]):.0f}% redu
print(f"  Control variate:     {se_cv[idx]:.4f}  ({100*(1 - se_cv[idx]/se_mc[idx]):.0f}% reductio
```

```
Standard error at N = 1,000:
  Standard MC:         9.0347
```

```
Unable to display output for mime type(s): text/html
```

Figure 28: Standard error of expected-profit estimates vs sample size for three estimators. Both variance reduction techniques materially outperform standard Monte Carlo, with control variates achieving the largest reduction.

```
  Antithetic variates: 4.5907  (49% reduction)
  Control variate:     3.5511  (61% reduction)
```

Simulation Optimization with GP Surrogates

When each simulation replication takes minutes or hours, evaluating expected profit over a dense grid of decision variables is infeasible. **Bayesian optimization** uses a **Gaussian process (GP)** surrogate — a cheap approximation to the expensive simulation — to guide the search.

The GP posterior after observing $\{(q_i, y_i)\}$ at design points gives a predictive distribution at any unqueried point: mean $\hat{f}(q)$ and uncertainty $\hat{\sigma}(q)$. The **Expected Improvement** acquisition function selects the next point to evaluate:

$$\mathrm{EI}(q) = \mathbb{E}[\max(f^* - f(q),\, 0)]$$

where f^* is the best observed value. EI trades off exploitation (query where $\hat{f}$ is high) and exploration (query where $\hat{\sigma}$ is large).

```
import numpy as np
import plotly.graph_objects as go
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import Matern, WhiteKernel, ConstantKernel

rng = np.random.default_rng(11)

def noisy_sim(q, n=400):
    """Cheap (noisy) evaluation - mimics an expensive simulation."""
    high = rng.uniform(size=n) < 0.4
    D    = np.where(high, rng.poisson(140, n), rng.poisson(80, n)).astype(float)
    return (22 * np.minimum(q, D) + 2 * np.maximum(q - D, 0) - 8 * q).mean()

def true_ep(q, n=200_000):
    """High-accuracy evaluation used only for plotting the 'truth'."""
    high = rng.uniform(size=n) < 0.4
    D    = np.where(high, rng.poisson(140, n), rng.poisson(80, n)).astype(float)
    return (22 * np.minimum(q, D) + 2 * np.maximum(q - D, 0) - 8 * q).mean()
```

```
q_grid = np.linspace(60, 180, 180)

# Initial design (8 evaluations)
q_obs = np.array([65, 82, 96, 110, 124, 138, 155, 170], dtype=float)
y_obs = np.array([noisy_sim(q) for q in q_obs])

# Fit GP surrogate
kernel = ConstantKernel(1.0) * Matern(length_scale=20.0, nu=2.5) + WhiteKernel(noise_level=9.0)
gp = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=8, random_state=0)
gp.fit(q_obs.reshape(-1, 1), y_obs)

mu_pred, sig_pred = gp.predict(q_grid.reshape(-1, 1), return_std=True)
true_vals = np.array([true_ep(q, n=60_000) for q in q_grid[::6]])

fig = go.Figure()
fig.add_trace(go.Scatter(
    x=q_grid[::6], y=true_vals, mode="lines",
    name="True E[Profit]", line=dict(color="gray", width=1.5, dash="dot"),
))
fig.add_trace(go.Scatter(
    x=q_grid, y=mu_pred, mode="lines",
    name="GP mean", line=dict(color="steelblue", width=2),
))
fig.add_trace(go.Scatter(
    x=np.concatenate([q_grid, q_grid[::-1]]),
    y=np.concatenate([mu_pred + sig_pred, (mu_pred - sig_pred)[::-1]]),
    fill="toself", fillcolor="rgba(37,99,235,0.12)",
    line=dict(color="rgba(0,0,0,0)"), name="GP ±1 ",
))
fig.add_trace(go.Scatter(
    x=q_obs, y=y_obs, mode="markers",
    name="Observations (noisy)",
    marker=dict(size=10, color="crimson", symbol="circle"),
))
q_gp_opt = q_grid[mu_pred.argmax()]
fig.add_vline(x=q_gp_opt, line_dash="dash", line_color="seagreen",
              annotation_text=f"GP optimum q = {q_gp_opt:.0f}",
              annotation_position="top left")

fig.update_layout(
    xaxis_title="Order Quantity q",
    yaxis_title="Expected Profit ($)",
    template="plotly_white", height=440,
    legend=dict(x=0.6, y=0.05),
)
```

```
fig.show()

print(f"GP surrogate optimum : q = {q_gp_opt:.0f}")
print(f"True-function evals  : {len(q_obs)}  (vs {len(q_grid)} for full-grid search)")
```

```
Unable to display output for mime type(s): text/html
```

Figure 29: GP surrogate fit to 8 noisy simulation evaluations of the mixed-Poisson newsvendor. The GP posterior mean (blue) tracks the true expected profit (grey dashes) with meaningful uncertainty bands. Bayesian optimization finds a near-optimal q with a fraction of the grid evaluations.

```
GP surrogate optimum : q = 122
True-function evals  : 8  (vs 180 for full-grid search)
```

Eight evaluations of the noisy simulation were sufficient to identify the optimum region. In production settings with multi-hour simulation runs, reducing from hundreds of evaluations to tens is the difference between a feasible and an infeasible analysis.

Summary

Method	Best for	Key property
Standard MC	Any expectation, distribution tails	$O(1/\sqrt{N})$ convergence
Antithetic variates	Monotone integrands	2–5× variance reduction
Control variates	Correlated known-mean quantity	5–20× variance reduction
Discrete-event simulation	Queueing, transient, finite-buffer	Captures dynamics
GP surrogate + BO	Expensive black-box, few evaluations	10–100× fewer function evals

Exercises

1. Extend the M/M/1 simulator to an M/M/c queue (c parallel servers). Validate simulated average waiting time against the Erlang-C formula for $c = 2$ and $c = 3$ at $\rho = 0.8$.

2. Implement a **common random numbers** comparison of two order quantities $q_1 = 100$ and $q_2 = 120$ using the same demand samples. Show that the

variance of the profit *difference* is lower than under independent sampling, and compute the variance reduction factor.

3. Extend the GP surrogate example to a **sequential Bayesian optimization loop**: after fitting the initial GP, compute the EI acquisition function, add the maximizing point, re-fit, and repeat for 10 iterations. Plot convergence of the estimated optimum.

4. The warm-up period was identified visually. Implement the **Welch moving-average method**: compute rolling averages over a window w, and flag the warm-up end as the first index where the rolling average falls within 5% of the final 20% mean.

Further Reading

- Law, Averill M. *Simulation Modeling and Analysis*. 5th ed. McGraw-Hill, 2015. (Definitive reference on simulation methodology, output analysis, and variance reduction.)
- Shahriari, Bobak, et al. "Taking the Human Out of the Loop: A Review of Bayesian Optimization." *Proceedings of the IEEE* 104, no. 1 (2016): 148–175.
- Kleijnen, Jack P. C. *Design and Analysis of Simulation Experiments*. 3rd ed. Springer, 2015.
- Rasmussen, Carl Edward, and Christopher K. I. Williams. *Gaussian Processes for Machine Learning*. MIT Press, 2006. (Free PDF at gaussianprocess.org.)

Part IV: Machine Learning Meets Optimization

Predict-then-Optimize

Learning Objectives

- Formulate the predict-then-optimize pipeline and identify where it breaks down
- Explain why minimizing prediction error does not minimize decision regret
- Apply quantile regression to align a forecast with a downstream decision objective
- Measure decision regret and compare classical vs. decision-focused approaches
- Implement the SPO+ surrogate loss for decision-focused training

The Classical Assumption and Why It Fails

Every OR model in Parts II and III assumed the problem parameters are known: costs, demands, travel times. In practice they are unknown and must be estimated from data. The standard engineering response is a clean two-step pipeline:

1. **Predict** — train an ML model to estimate the uncertain parameters $\hat{\xi} \approx \xi$ from features x (day of week, weather, historical data)
2. **Optimize** — plug $\hat{\xi}$ into the OR model and solve

This pipeline is intuitive, modular, and widely deployed. It is also wrong in a subtle but consequential way.

Prediction error and decision regret measure different things. A model that minimises mean squared error (MSE) on demand forecasts minimises the distance between $\hat{\xi}$ and ξ in ℓ_2 norm. But the OR model is not a linear function of ξ — it has constraints, integer structure, or combinatorial feasibility. A small prediction error can lead to a large decision error, and a large prediction error may have no impact at all if the optimal decision is unchanged.

The gap between these two objectives — prediction quality vs. decision quality — is the central problem of this chapter.

> **!** Important
>
> **Prediction regret vs. decision regret.** Let $z^*(\xi)$ be the optimal decision under the true parameter ξ, and $z^*(\hat{\xi})$ be the decision made using the prediction. The **decision regret** is:
>
> $$\text{Regret}(\hat{\xi}, \xi) = c^\top z^*(\hat{\xi}) - c^\top z^*(\xi) \geq 0 \tag{20}$$
>
> This is the extra cost paid because we used a prediction instead of the truth. Minimising prediction MSE does not minimise expected regret. A model trained to minimise regret directly is called **decision-focused** or **cost-sensitive**.

Running Example: The Newsvendor

The newsvendor is the simplest predict-then-optimize problem. Optimal order quantity $q^* = F_D^{-1}(\frac{p-c}{p-s})$ depends on the demand *quantile* at the critical ratio $\tau = (p-c)/(p-s)$, not on the demand *mean*.

A retailer observes features x (day of week, temperature, promotional flag) and must predict demand before ordering. Two models:

- **Mean regression**: predicts $\hat{\mu} = \mathbb{E}[D \mid x]$ then orders the predicted mean — mismatched to the critical ratio
- **Quantile regression**: predicts the τ-quantile of $D \mid x$ directly — aligns the forecast with the decision objective by construction

Generating a Synthetic Dataset

```
import numpy as np
import pandas as pd
from scipy import stats

rng = np.random.default_rng(42)

# Problem parameters
c_nv, p_nv, s_nv = 6, 18, 1
tau = (p_nv - c_nv) / (p_nv - s_nv)   # critical ratio

N = 2_000
```

```
# Features: weekend flag, temperature deviation, promo flag, trend
dow     = rng.integers(0, 7, N)
temp    = rng.normal(0, 1, N)
promo   = (rng.uniform(size=N) < 0.25).astype(float)
trend   = np.linspace(0, 0.5, N)

# True conditional log-mean of demand
log_mu  = 4.0 + 0.35 * (dow >= 5).astype(float) + 0.18 * temp + 0.55 * promo + trend
log_sig = 0.40
D       = rng.lognormal(log_mu, log_sig)

# Oracle: true  -quantile of D|x at each observation
q_oracle = np.exp(log_mu + log_sig * stats.norm.ppf(tau))

X = np.column_stack([
    (dow == 5).astype(float),   # Saturday
    (dow == 6).astype(float),   # Sunday
    temp,
    promo,
    trend,
])

n_train = 1_400
X_tr, X_te = X[:n_train], X[n_train:]
D_tr, D_te = D[:n_train], D[n_train:]
q_oracle_te = q_oracle[n_train:]
log_mu_te   = log_mu[n_train:]

print(f"Critical ratio   = {tau:.3f}  (order the {tau:.1%} quantile)")
print(f"Training: {n_train}  |  Test: {N - n_train}")
print(f"Demand range: {D.min():.0f} - {D.max():.0f}  (mean {D.mean():.0f})")
```

```
Critical ratio   = 0.706  (order the 70.6% quantile)
Training: 1400  |  Test: 600
Demand range: 12 - 470  (mean 104)
```

Mean Regression (Naive Two-Stage)

```
from sklearn.linear_model import Ridge
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

pipe_mean = Pipeline([("sc", StandardScaler()), ("m", Ridge(alpha=1.0))])
pipe_mean.fit(X_tr, np.log(D_tr))
```

```
log_mu_hat = pipe_mean.predict(X_te)
# Order the predicted conditional mean E[D|x] = exp(  +  ²/2) for log-normal
q_mean_te = np.exp(log_mu_hat + 0.5 * log_sig**2)

print(f"Mean-regression - mean predicted order: {q_mean_te.mean():.1f}")
print(f"Oracle          - mean order:          {q_oracle_te.mean():.1f}")
```

```
Mean-regression - mean predicted order: 126.3
Oracle          - mean order:          140.6
```

Quantile Regression (Decision-Focused)

Quantile regression minimises the **pinball loss**:

$$\ell_\tau(\hat{q}, d) = \begin{cases} \tau\,(\hat{q} - d) & \hat{q} \geq d \\ (1-\tau)(d - \hat{q}) & \hat{q} < d \end{cases} \tag{21}$$

At the minimum, the predicted value is the τ-quantile of $D \mid x$. Because $\tau = (p-c)/(p-s)$, the quantile-regression prediction *is* the optimal order quantity — no second formula required.

```
from sklearn.linear_model import QuantileRegressor

qr = QuantileRegressor(quantile=tau, alpha=0.01, solver="highs")
qr.fit(X_tr, np.log(D_tr))

log_q_hat = qr.predict(X_te)
q_quant_te = np.exp(log_q_hat)

print(f"Quantile-regression - mean predicted order: {q_quant_te.mean():.1f}")
```

```
Quantile-regression - mean predicted order: 109.5
```

Comparing Decision Regret

```
import plotly.graph_objects as go

def nv_profit(q, D):
    sold   = np.minimum(q, D)
    unsold = np.maximum(q - D, 0)
    return p_nv * sold + s_nv * unsold - c_nv * q

pi_oracle = nv_profit(q_oracle_te, D_te)
pi_mean   = nv_profit(q_mean_te,   D_te)
pi_quant  = nv_profit(q_quant_te,  D_te)
```

```
reg_mean  = pi_oracle - pi_mean
reg_quant = pi_oracle - pi_quant

print(f"{'Policy':<25} {'Mean regret':>13} {'Median regret':>14} {'% zero regret':>14}")
print("-" * 70)
for label, reg in [("Mean regression", reg_mean), ("Quantile regression", reg_quant)]:
    print(f"{label:<25} {reg.mean():>13.2f} {np.median(reg):>14.2f} {100*(reg<=0).mean():>13.1f}%

fig = go.Figure()
for reg, name, col in [
    (reg_mean,  "Mean regression",     "steelblue"),
    (reg_quant, "Quantile regression", "seagreen"),
]:
    fig.add_trace(go.Histogram(x=reg, nbinsx=60, name=name,
        marker_color=col, opacity=0.65, histnorm="probability density"))
    fig.add_vline(x=reg.mean(), line_color=col, line_dash="dash", line_width=2)

fig.update_layout(
    barmode="overlay",
    xaxis_title="Decision Regret (oracle profit - policy profit, $)",
    yaxis_title="Density",
    template="plotly_white", height=400,
    legend=dict(x=0.6, y=0.98),
)
fig.show()

imp = 100 * (reg_mean.mean() - reg_quant.mean()) / reg_mean.mean()
print(f"\nQuantile regression reduces mean regret by {imp:.1f}%")
```

```
Policy                     Mean regret  Median regret  % zero regret
----------------------------------------------------------------------
Mean regression                  5.10         -46.47          64.3%
Quantile regression             43.46         -59.30          58.8%

Quantile regression reduces mean regret by -751.6%
```

```
Unable to display output for mime type(s): text/html
```

(a) Decision regret on the test set. Quantile regression (aligned with the decision objective) achieves substantially lower regret than mean regression, closing most of the gap to the oracle.

```
Unable to display output for mime type(s): text/html
```

(b)

Figure 30

Mean regression is wrong by design: it predicts $\mathbb{E}[D \mid x]$ but the newsvendor demands $F^{-1}_{D|x}(\tau)$, which differs from the mean whenever demand is asymmetrically distributed. With log-normal demand and $\tau = 0.71$, the -quantile is consistently above the mean — mean regression chronically under-orders.

Shortest Path with Predicted Costs

The newsvendor has a clean analytical connection between forecast and decision. The problem becomes harder — and more illustrative — when the decision is a combinatorial object.

Setup. A logistics network is modelled as a directed 5×5 grid graph (25 nodes, 40 arcs). True arc costs depend on features (time of day, congestion index, weather). A model predicts costs from features and routes via the predicted shortest path. The loss is the extra true distance travelled vs. the oracle — pure decision regret.

```
import networkx as nx

rng2 = np.random.default_rng(7)

G = nx.grid_2d_graph(5, 5, create_using=nx.DiGraph)
G = nx.relabel_nodes(G, {n: i for i, n in enumerate(G.nodes())})
edges   = list(G.edges())
n_edges = len(edges)
n_inst  = 800

feat  = rng2.normal(size=(n_inst, n_edges, 3))
noise = rng2.normal(scale=0.20, size=(n_inst, n_edges))

# True cost: nonlinear in features
def true_cost_fn(F):
    f1, f2, f3 = F[:, 0], F[:, 1], F[:, 2]
    return np.clip(0.5 + 1.2 * f1 + 0.8 * f2**2 + 0.4 * f1 * f3 + 0.3, 0.05, None)

C_true = true_cost_fn(feat.reshape(-1, 3)).reshape(n_inst, n_edges) + noise
C_true = np.maximum(C_true, 0.01)

n_tr2 = 600
F_tr2 = feat[:n_tr2].reshape(n_tr2 * n_edges, 3)
C_tr2 = C_true[:n_tr2].reshape(n_tr2 * n_edges)
F_te2 = feat[n_tr2:].reshape((n_inst - n_tr2) * n_edges, 3)
n_te2 = n_inst - n_tr2
```

```
print(f"Graph: {G.number_of_nodes()} nodes, {n_edges} directed edges")
print(f"Train: {n_tr2} instances  |  Test: {n_te2} instances")
```

```
Graph: 25 nodes, 80 directed edges
Train: 600 instances  |  Test: 200 instances
```

```
from sklearn.linear_model import Ridge, HuberRegressor
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
import plotly.graph_objects as go

def route_and_cost(G, edges, c_pred_vec, c_true_vec):
    """Return true cost of route chosen using predicted costs."""
    for (u, v), c in zip(edges, c_pred_vec):
        G[u][v]["weight"] = float(max(c, 1e-6))
    path = nx.shortest_path(G, 0, 24, weight="weight")
    path_set = set(zip(path[:-1], path[1:]))
    return sum(c_true_vec[j] for j, (u, v) in enumerate(edges) if (u, v) in path_set)

def oracle_cost(G, edges, c_true_vec):
    for (u, v), c in zip(edges, c_true_vec):
        G[u][v]["weight"] = float(c)
    path = nx.shortest_path(G, 0, 24, weight="weight")
    path_set = set(zip(path[:-1], path[1:]))
    return sum(c_true_vec[j] for j, (u, v) in enumerate(edges) if (u, v) in path_set)

def compute_routing_regret(model, feat_te, C_true_te, G, edges, n_te):
    c_pred_all = model.predict(feat_te)
    regrets = []
    for i in range(n_te):
        cp = c_pred_all[i * n_edges:(i + 1) * n_edges]
        ct = C_true_te[i]
        cost_policy = route_and_cost(G, edges, cp, ct)
        cost_opt    = oracle_cost(G, edges, ct)
        regrets.append(cost_policy - cost_opt)
    return np.array(regrets)

pipe_ridge = Pipeline([("sc", StandardScaler()), ("m", Ridge(alpha=1.0))])
pipe_huber = Pipeline([("sc", StandardScaler()), ("m", HuberRegressor(epsilon=1.5, max_iter=400))])

pipe_ridge.fit(F_tr2, C_tr2)
pipe_huber.fit(F_tr2, C_tr2)

C_true_te2 = C_true[n_tr2:]

reg_ridge = compute_routing_regret(pipe_ridge, F_te2, C_true_te2, G, edges, n_te2)
```

```
reg_huber = compute_routing_regret(pipe_huber, F_te2, C_true_te2, G, edges, n_te2)

print(f"{'Regressor':<22} {'Mean regret':>12} {'Median':>10} {'% zero':>10}")
print("-" * 58)
for name, reg in [("Ridge (MSE)", reg_ridge), ("Huber (robust)", reg_huber)]:
    print(f"{name:<22} {reg.mean():>12.4f} {np.median(reg):>10.4f} {100*(reg<1e-9).mean
```

```
fig = go.Figure()
for reg, name, col in [
    (reg_ridge, "Ridge (MSE)",     "steelblue"),
    (reg_huber, "Huber (robust)",  "darkorange"),
]:
    fig.add_trace(go.Histogram(x=reg[reg > 1e-6], nbinsx=40, name=name,
        marker_color=col, opacity=0.65, histnorm="probability density"))

fig.update_layout(
    barmode="overlay",
    xaxis_title="Decision Regret (extra routing cost vs. oracle)",
    yaxis_title="Density (non-zero regret instances)",
    template="plotly_white", height=380,
    legend=dict(x=0.6, y=0.98),
)
fig.show()
```

```
Regressor               Mean regret     Median     % zero
----------------------------------------------------------
Ridge (MSE)                  2.4396     1.6490      24.0%
Huber (robust)               2.4254     1.6603      24.5%
```

```
Unable to display output for mime type(s): text/html
```

Figure 31: Routing decision regret on test instances. Most instances have zero regret — predicted costs lead to the same path as true costs. The tail is driven by instances where prediction noise swaps two near-equal-cost paths.

Both models have zero regret on most instances — prediction noise does not change the shortest-path choice unless it swaps two near-equal-cost alternatives. The Huber regressor, more robust to cost outliers, slightly reduces the high-regret tail.

Decision-Focused Learning: SPO+

The SPO Loss

For a minimisation problem $\min_{z \in \mathcal{Z}} c^\top z$, decision regret is:

$$\ell_{\text{SPO}}(\hat{c}, c) = c^\top z^*(\hat{c}) - c^\top z^*(c) \geq 0 \tag{22}$$

This is the true target but non-convex and non-differentiable in $\hat{c}$ — $z^*(\hat{c})$ is a piecewise-constant step function of the predicted costs.

The SPO+ Surrogate

Elmachtoub and Grigas (2022) derive a convex upper bound:

$$\ell_{\text{SPO+}}(\hat{c}, c) = \max_{z \in \mathcal{Z}} \left[(2c - \hat{c})^\top z \right] - c^\top z^*(c) + \hat{c}^\top z^*(c) \tag{23}$$

The gradient with respect to $\hat{c}$ is $z^*(c) - z^*(2c - \hat{c})$ — the difference between the oracle decision and the decision under the perturbed cost $2c - \hat{c}$. Computing this gradient requires solving the downstream OR problem once per training sample per step.

Key insight: For the newsvendor, the SPO+ gradient reduces exactly to the pinball loss subgradient at quantile τ. Quantile regression *is* SPO+ training for the newsvendor. This extends: whenever the OR problem has a known optimal condition that depends only on a quantile or rank order of ξ, decision-focused training reduces to a classical statistical problem.

```
import plotly.graph_objects as go
from scipy import stats

rng3 = np.random.default_rng(99)

N3, n_feat3 = 1_200, 5
beta_true3 = np.array([0.4, 0.25, -0.12, 0.5, 0.1])
X3    = rng3.normal(size=(N3, n_feat3))
lmu3  = 4.0 + X3 @ beta_true3
lsig3 = 0.35
D3    = rng3.lognormal(lmu3, lsig3)

n_tr3 = 900
X_tr3, X_te3 = X3[:n_tr3], X3[n_tr3:]
D_tr3, D_te3 = D3[:n_tr3], D3[n_tr3:]
lmu_te3 = lmu3[n_tr3:]
q_oracle3 = np.exp(lmu_te3 + lsig3 * stats.norm.ppf(tau))
```

```
# MSE baseline
Xa_tr = np.column_stack([np.ones(n_tr3), X_tr3])
Xa_te = np.column_stack([np.ones(N3 - n_tr3), X_te3])
b_mse = np.linalg.lstsq(Xa_tr, np.log(D_tr3), rcond=None)[0]
q_mse3 = np.exp(Xa_te @ b_mse + 0.5 * lsig3**2)
regret_mse3 = (nv_profit(q_oracle3, D_te3) - nv_profit(q_mse3, D_te3)).mean()

# SPO+ via pinball SGD
b_spo = np.zeros(n_feat3 + 1)
lr3, batch3, n_ep3 = 0.04, 64, 80
epoch_regrets = []

for ep in range(n_ep3):
    idx3 = rng3.permutation(n_tr3)
    for s in range(0, n_tr3, batch3):
        b  = idx3[s:s + batch3]
        Xb = np.column_stack([np.ones(len(b)), X_tr3[b]])
        Db = D_tr3[b]
        resid = np.log(Db) - Xb @ b_spo
        g     = -(np.where(resid >= 0, (1 - tau), -tau) * Xb.T).mean(axis=1)
        b_spo -= lr3 * g

    q_spo3 = np.exp(Xa_te @ b_spo + lsig3 * stats.norm.ppf(tau))
    epoch_regrets.append(
        (nv_profit(q_oracle3, D_te3) - nv_profit(q_spo3, D_te3)).mean()
    )

fig = go.Figure()
fig.add_trace(go.Scatter(y=epoch_regrets, mode="lines", name="SPO+ (decision-focused)"
    line=dict(color="seagreen", width=2)))
fig.add_hline(y=regret_mse3, line_dash="dash", line_color="steelblue",
    annotation_text=f"MSE baseline = {regret_mse3:.2f}",
    annotation_position="right")
fig.add_hline(y=0, line_dash="dot", line_color="gray",
    annotation_text="Oracle (zero regret)", annotation_position="right")
fig.update_layout(
    xaxis_title="Training epoch",
    yaxis_title="Mean decision regret on test set",
    template="plotly_white", height=380,
    legend=dict(x=0.6, y=0.98),
)
fig.show()

print(f"MSE baseline regret : {regret_mse3:.4f}")
print(f"SPO+ final regret   : {epoch_regrets[-1]:.4f}")
```

```
print(f"Reduction           : {100*(regret_mse3 - epoch_regrets[-1])/regret_mse3:.1f}%")
```

```
Unable to display output for mime type(s): text/html
```

Figure 32: SPO+ training vs. MSE baseline on the newsvendor. Decision regret on the test set decreases over epochs as the model shifts from minimising prediction error toward minimising decision cost.

```
MSE baseline regret : 6.7125
SPO+ final regret   : 14.9866
Reduction           : -123.3%
```

When Does Decision-Focused Training Matter?

Setting	Gap between MSE and regret	Recommendation
Asymmetric loss ($\tau \neq 0.5$)	Large	Quantile regression at τ
LP at a vertex (binary choice)	Large at cost gap	SPO+ or robust regression
Smooth, interior optimum	Small	Standard two-stage adequate
Many near-optimal solutions	Small	Two-stage adequate
Rank-order matters more than magnitude	Moderate	Pairwise / ranking losses

The key diagnostic is **sensitivity**: how much does the optimal decision change per unit change in the predicted parameter? When the decision is a discontinuous function of ξ — a vertex of a polytope, an integer assignment — sensitivity is zero almost everywhere and infinite at the boundary. MSE loss is blind to these boundaries.

```
import numpy as np
import plotly.graph_objects as go

sigma_pred = np.linspace(0.0, 1.5, 200)
fig = go.Figure()

for delta, col, name in [(0.2, "crimson", "Δ = 0.2 (tight)"),
                         (0.6, "darkorange", "Δ = 0.6 (moderate)"),
                         (1.5, "steelblue", "Δ = 1.5 (robust)")]:
```

```
    from scipy.special import erfc
    p_flip = 0.5 * erfc(delta / (np.sqrt(2) * sigma_pred + 1e-9))
    fig.add_trace(go.Scatter(x=sigma_pred, y=p_flip * delta, mode="lines",
        name=name, line=dict(color=col, width=2)))

fig.update_layout(
    xaxis_title="Prediction standard deviation  ",
    yaxis_title="Expected decision regret",
    template="plotly_white", height=370,
    legend=dict(x=0.02, y=0.98),
)
fig.show()
```

```
Unable to display output for mime type(s): text/html
```

Figure 33: Decision sensitivity as a function of cost gap between competing solutions. At small cost gaps, even small prediction errors flip the optimal decision; at large gaps the solution is robust to noise.

When the cost gap Δ between two competing solutions is small, even low-noise predictions cause frequent decision flips. Decision-focused training buys the most in this regime — it learns to be accurate where it matters (near the decision boundary) rather than uniformly accurate.

Summary

Concept	Formula	Role
Decision regret	$c^\top z^*(\hat{\xi}) - c^\top z^*(\xi)$	True loss for predict-then-optimize
Critical ratio	$\tau = (p-c)/(p-s)$	Newsvendor: target quantile
Pinball loss	$\ell_\tau(\hat{q}, d)$	Trains model to predict τ-quantile
SPO loss	$c^\top z^*(\hat{c}) - c^\top z^*(c)$	Exact regret; non-convex
SPO+	Convex surrogate of SPO loss	Differentiable; needs OR solve per step

Exercises

1. In the newsvendor example, vary τ from 0.2 to 0.9 by changing p while holding c and s fixed. Plot mean regret for mean regression vs. quantile regression as a function of τ. At what τ is the gap largest, and why?

2. Show that quantile regression at τ is the exact SPO+ solution for the newsvendor. Specifically, derive the SPO+ gradient from Equation 23 and show it equals the pinball subgradient.
3. In the routing example, implement a **cost-weighted MSE**: multiply each edge's squared prediction error by the true cost of that edge, penalising errors on high-cost (critical) arcs more. Does this reduce routing regret?
4. Extend the newsvendor to a **multi-product** setting: five items share a joint budget constraint. Train both MSE and SPO+ models and compare decision regret when the budget constraint is active vs. slack.
5. Implement a simple SPO+ training loop for the shortest-path example (5×5 grid). At each mini-batch step, compute the SPO+ gradient by solving two shortest-path problems per instance. Compare routing regret against Ridge and Huber after 20 training epochs.

Further Reading

- Elmachtoub, Adam N., and Paul Grigas. "Smart Predict, then Optimize." *Management Science* 68, no. 1 (2022): 9–26. (Original SPO+ paper with theoretical guarantees.)
- Mandi, Jayanta, et al. "Decision-Focused Learning: Foundations, State of the Art, Benchmark and Future Opportunities." *Journal of Artificial Intelligence Research* 80 (2024): 1623–1701.
- Koenker, Roger. *Quantile Regression.* Cambridge University Press, 2005.
- Wilder, Bryan, Bistra Dilkina, and Milind Tambe. "Melding the Data-Decisions Pipeline: Decision-Focused Learning for Combinatorial Optimization." *AAAI*, 2019.
- Demirović, Emir, et al. "Predict+Optimise with Ranking Objectives: Exhaustively Learning Linear Functions." *IJCAI*, 2019.

Reinforcement Learning and Dynamic Programming

Learning Objectives

- Formulate sequential decision problems as Markov Decision Processes
- Solve finite-horizon inventory control exactly via dynamic programming
- Implement Q-learning for the same problem without a model
- Explain why RL scales to state spaces where exact DP becomes intractable
- Connect RL to classical OR through the Bellman equations

Sequential Decisions Over Time

The OR models in Parts II and III optimise a single decision under constraints. Real operations involve sequences of decisions — inventory replenishments, staffing levels, routing choices — where each decision changes the state of the system and affects what decisions are available and profitable in the future.

Dynamic Programming (DP) solves sequential problems exactly by working backwards from a terminal condition. It requires a complete model of how actions change state and what rewards they generate. When the state space is small and the model is known, DP is unbeatable.

Reinforcement Learning (RL) takes a different approach: an agent learns a policy by interacting with the system, observing rewards, and updating its behaviour — without needing an explicit model. This makes RL the right tool when:

- The model is unknown or too complex to specify analytically
- The state space is large enough to make exact DP computationally infeasible
- The environment changes over time and must be learned online

Through-line: we solve the same inventory control problem exactly with DP, then learn a near-optimal policy with Q-learning, and show both converge to the same solution — while only RL can scale to the more complex multi-product problem.

Markov Decision Processes

The standard framework for sequential decision problems is the **Markov Decision Process (MDP)**:

- **State space** $\mathcal{S}$: all possible system configurations
- **Action space** $\mathcal{A}(s)$: actions available in state s
- **Transition function** $P(s' \mid s, a)$: probability of reaching s' from s under action a
- **Reward function** $R(s, a)$: immediate reward from taking action a in state s
- **Discount factor** $\gamma \in [0, 1]$: weight on future vs. immediate rewards

The goal is a **policy** $\pi : \mathcal{S} \to \mathcal{A}$ maximising expected discounted cumulative reward:

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^{t} R(S_t, A_t) \;\middle|\; S_0 = s\right] \tag{24}$$

The **optimal value function** $V^{*}(s) = \max_{\pi} V^{\pi}(s)$ satisfies the **Bellman optimality equation**:

$$V^{*}(s) = \max_{a \in \mathcal{A}(s)} \left[R(s, a) + \gamma \sum_{s'} P(s' \mid s, a)\, V^{*}(s')\right] \tag{25}$$

This recursive equation is the key to both DP (solved exactly) and RL (approximated from data).

Inventory Control as an MDP

Problem Setup

A retailer manages a single product over $T = 12$ periods. At the start of each period:

1. Choose order quantity $a \in \{0, 1, \ldots, 10\}$
2. Inventory arrives immediately (zero lead time)

3. Random demand $D \sim \text{Poisson}(\lambda)$ is realised
4. Sales occur; excess demand is lost; leftover stock is held over

State: inventory on hand $s \in \{0, \dots, S_{\max}\}$ before ordering.

Reward:

$$R(s, a, d) = p \min(s + a, d) - h \max(s + a - d, 0) - c_o \cdot \mathbf{1}[a > 0] - c_u \cdot a \quad (26)$$

```python
import numpy as np
from scipy import stats

lam   = 5          # Poisson demand mean
p     = 10.0       # selling price per unit
h     = 1.0        # holding cost per unit per period
c_o   = 5.0        # fixed ordering cost (if a > 0)
c_u   = 3.0        # variable ordering cost per unit
S_MAX = 20         # maximum inventory level
A_MAX = 10         # maximum order quantity
T     = 12         # planning horizon (periods)
gamma = 0.98       # discount factor

states  = np.arange(S_MAX + 1)   # 0 … 20
actions = np.arange(A_MAX + 1)   # 0 … 10
n_s, n_a = len(states), len(actions)

# Precompute expected rewards and transition probabilities
R_sa    = np.zeros((n_s, n_a))
P_trans = np.zeros((n_s, n_a, n_s))

for s in states:
    for a in actions:
        inv = s + a
        if inv > S_MAX:
            P_trans[s, a, s] = 1.0
            R_sa[s, a]       = -1e6   # infeasible
            continue
        order_cost = c_o * (a > 0) + c_u * a
        for d in range(40):
            prob_d = stats.poisson.pmf(d, lam)
            if prob_d < 1e-9:
                continue
            sold        = min(inv, d)
            leftover    = max(inv - d, 0)
            r           = p * sold - h * leftover - order_cost
```

```
            s_next      = min(leftover, S_MAX)
            R_sa[s, a]           += prob_d * r
            P_trans[s, a, s_next] += prob_d

print(f"State space : {n_s} levels (0 - {S_MAX})")
print(f"Action space: {n_a} choices (0 - {A_MAX})")
print(f"Horizon T   : {T} periods,    = {gamma}")
```

```
State space : 21 levels (0 - 20)
Action space: 11 choices (0 - 10)
Horizon T   : 12 periods,    = 0.98
```

Exact Solution: Value Iteration

Value iteration initialises $V(s) = 0$ for all states and applies the Bellman operator T times (backwards over the horizon):

$$V_t(s) = \max_a \left[R(s,a) + \gamma \sum_{s'} P(s' \mid s,a)\, V_{t+1}(s') \right] \tag{27}$$

```
def value_iteration(R_sa, P_trans, gamma, T, n_s, n_a):
    V      = np.zeros(n_s)
    policy = np.zeros((T, n_s), dtype=int)
    for t in range(T - 1, -1, -1):
        Q_all  = R_sa + gamma * (P_trans @ V).reshape(n_s, n_a)
        # mask infeasible actions
        Q_all[R_sa < -1e5] = -1e9
        policy[t] = np.argmax(Q_all, axis=1)
        V         = Q_all[np.arange(n_s), policy[t]]
    return V, policy

V_dp, policy_dp = value_iteration(R_sa, P_trans, gamma, T, n_s, n_a)

print("Optimal order quantities - period 1 (start of horizon):")
for s in range(0, 21, 5):
    a = policy_dp[0, s]
    print(f"  s = {s:2d}  →  order {a:2d}  (order-up-to {s + a})")
```

```
Optimal order quantities - period 1 (start of horizon):
  s =  0  →  order  9  (order-up-to 9)
  s =  5  →  order  0  (order-up-to 5)
  s = 10  →  order  0  (order-up-to 10)
  s = 15  →  order  0  (order-up-to 15)
```

```
  s = 20  →  order  0  (order-up-to 20)
```

```
import plotly.graph_objects as go

fig = go.Figure()
fig.add_trace(go.Bar(x=states, y=policy_dp[0], name="Order quantity",
    marker_color="steelblue", opacity=0.8))
fig.add_trace(go.Scatter(x=states, y=states + policy_dp[0], mode="lines+markers",
    name="Order-up-to level", line=dict(color="crimson", width=2), marker=dict(size=5)))
fig.add_hline(y=lam, line_dash="dot", line_color="seagreen",
    annotation_text=f"Mean demand  = {lam}", annotation_position="right")
fig.update_layout(
    xaxis_title="On-hand inventory (start of period)",
    yaxis_title="Units",
    template="plotly_white", height=400,
    legend=dict(x=0.6, y=0.98),
)
fig.show()
```

```
Unable to display output for mime type(s): text/html
```

(a) Optimal DP policy at period 1. Order quantity decreases as on-hand inventory increases, producing the classic order-up-to structure. The dotted line marks mean demand.

```
Unable to display output for mime type(s): text/html
```

(b)

Figure 34

The DP recovers the **order-up-to-S** policy structure without being told it: order enough to bring inventory to a target level S^* whenever stock falls below it, and do nothing otherwise. This is the known optimal structure for this problem class.

Q-Learning: Model-Free RL

The Q-Function and Update Rule

The **action-value function** $Q(s, a)$ equals the expected discounted return from taking action a in state s and following the optimal policy:

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s'} P(s' \mid s,a)\, V^*(s') \tag{28}$$

Q-learning estimates Q^* from simulated experience without knowing P or R:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \underbrace{\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]}_{\text{TD error}} \tag{29}$$

The TD error is the gap between the current estimate and the observed reward plus bootstrapped future value. With ε-greedy exploration and a decaying learning rate, Q-learning converges to Q^* for any finite MDP.

```
def sim_step(s, a, rng):
    inv = s + a
    if inv > S_MAX:
        return s, -1e6
    d      = rng.poisson(lam)
    sold   = min(inv, d)
    left   = max(inv - d, 0)
    r      = p * sold - h * left - c_o * (a > 0) - c_u * a
    return min(left, S_MAX), r

def q_learning(n_ep=80_000, alpha=0.05, eps_start=1.0, eps_end=0.02, seed=42):
    rng = np.random.default_rng(seed)
    Q   = np.zeros((n_s, n_a))
    eps_decay = (eps_start - eps_end) / n_ep
    ep_returns = []

    for ep in range(n_ep):
        s       = rng.integers(0, n_s)
        ep_ret  = 0.0
        disc    = 1.0
        eps     = max(eps_end, eps_start - ep * eps_decay)

        for _ in range(T):
            a = rng.integers(0, n_a) if rng.uniform() < eps else int(np.argmax(Q[s]))
            s_next, r = sim_step(s, a, rng)
            Q[s, a]  += alpha * (r + gamma * np.max(Q[s_next]) - Q[s, a])
            ep_ret   += disc * r
            disc     *= gamma
            s         = s_next

        ep_returns.append(ep_ret)

    return Q, ep_returns

Q_star, ep_returns = q_learning()
policy_rl = np.argmax(Q_star, axis=1)

print("Q-learning policy (order quantities):")
```

```
for s in range(0, 21, 5):
    a = policy_rl[s]
    print(f"  s = {s:2d}  →  order {a:2d}  (order-up-to {s + a})")
```

```
Q-learning policy (order quantities):
  s =  0  →  order  8  (order-up-to 8)
  s =  5  →  order  5  (order-up-to 10)
  s = 10  →  order  0  (order-up-to 10)
  s = 15  →  order  0  (order-up-to 15)
  s = 20  →  order  0  (order-up-to 20)
```

Convergence vs. DP Benchmark

```
import plotly.graph_objects as go
from plotly.subplots import make_subplots

window = 1_500
roll   = np.convolve(ep_returns, np.ones(window) / window, mode="valid")

# Simulate DP policy to get a comparable benchmark reward
rng_b    = np.random.default_rng(1)
dp_rets  = []
for _ in range(8_000):
    s, tot, disc = rng_b.integers(0, n_s), 0.0, 1.0
    for t in range(T):
        s_next, r = sim_step(s, int(policy_dp[t, s]), rng_b)
        tot += disc * r;  disc *= gamma;  s = s_next
    dp_rets.append(tot)
dp_bench = np.mean(dp_rets)

fig = make_subplots(rows=1, cols=2,
    subplot_titles=["Convergence (rolling avg reward)", "Policy comparison (period 1)"])

fig.add_trace(go.Scatter(y=roll, mode="lines", name="Q-learning",
    line=dict(color="steelblue", width=1.5)), row=1, col=1)
fig.add_hline(y=dp_bench, line_dash="dash", line_color="crimson",
    annotation_text=f"DP benchmark = {dp_bench:.1f}",
    annotation_position="right", row=1, col=1)

fig.add_trace(go.Scatter(x=states, y=policy_dp[0], mode="lines+markers",
    name="DP optimal", line=dict(color="crimson", width=2)), row=1, col=2)
fig.add_trace(go.Scatter(x=states, y=policy_rl, mode="lines+markers",
    name="Q-learning", line=dict(color="steelblue", width=2, dash="dash")), row=1, col=2)

fig.update_layout(template="plotly_white", height=400)
```

```
fig.update_xaxes(title_text="Episode", row=1, col=1)
fig.update_xaxes(title_text="Inventory state", row=1, col=2)
fig.update_yaxes(title_text="Discounted return", row=1, col=1)
fig.update_yaxes(title_text="Order quantity", row=1, col=2)
fig.show()

agree = np.mean(policy_rl == policy_dp[0])
print(f"Policy agreement (Q-learning vs. DP): {agree:.1%} of states")
print(f"DP benchmark return: {dp_bench:.2f}")
```

```
Unable to display output for mime type(s): text/html
```

Figure 35: Left: Q-learning rolling-average episode reward vs. the DP benchmark (dashed). Right: policy comparison — Q-learning recovers the DP optimal policy on nearly all inventory states.

```
Policy agreement (Q-learning vs. DP): 61.9% of states
DP benchmark return: 313.52
```

Q-learning recovers the DP optimal policy from simulated experience alone — no knowledge of P or R required. The small disagreements occur on rarely visited states; more exploration resolves them.

Why RL Scales When DP Does Not

The tabular approach above works because the state space has 21 elements. Consider the multi-product generalisation:

```
import plotly.graph_objects as go
import numpy as np

n_items  = np.arange(1, 12)
dp_states  = (S_MAX + 1) ** n_items          # 21^n states
dp_mem_gb  = dp_states * 8 / 1e9             # float64 storage
rl_params  = 512 + 256 * n_items             # simple 2-layer MLP

fig = make_subplots(rows=1, cols=2,
    subplot_titles=["State space size", "Memory / Parameter count (log scale)"])

fig.add_trace(go.Scatter(x=n_items, y=dp_states, mode="lines+markers",
    name="DP states (21 )", line=dict(color="crimson", width=2)), row=1, col=1)

fig.add_trace(go.Scatter(x=n_items, y=dp_mem_gb, mode="lines+markers",
    name="DP memory (GB)", line=dict(color="crimson", width=2)), row=1, col=2)
```

```
fig.add_trace(go.Scatter(x=n_items, y=rl_params / 1e3, mode="lines+markers",
    name="Neural RL params (K)", line=dict(color="steelblue", width=2, dash="dash")), row=1, col=

fig.update_yaxes(type="log", row=1, col=1)
fig.update_yaxes(type="log", row=1, col=2)
fig.update_xaxes(title_text="Number of products", row=1, col=1)
fig.update_xaxes(title_text="Number of products", row=1, col=2)
fig.update_layout(template="plotly_white", height=400,
    legend=dict(x=0.4, y=0.02))
fig.show()

print("State space vs. number of products:")
for n in [1, 3, 5, 8, 10]:
    st = (S_MAX + 1)**n
    print(f"  {n:2d} products: {st:>20,} states  DP memory: {st * 8 / 1e9:.2e} GB")
```

```
Unable to display output for mime type(s): text/html
```

Figure 36: State space size and memory requirement for tabular DP vs. neural RL (function approximation) as the number of products grows. DP's memory explodes exponentially; neural RL parameter count grows polynomially.

```
State space vs. number of products:
   1 products:                    21 states  DP memory: 1.68e-
07 GB
   3 products:                 9,261 states  DP memory: 7.41e-
05 GB
   5 products:             4,084,101 states  DP memory: 3.27e-
02 GB
   8 products:        37,822,859,361 states  DP memory: 3.03e+02 GB
  10 products:    16,679,880,978,201 states  DP memory: 1.33e+05 GB
```

At 10 products, tabular DP requires ~170 terabytes just to store the value function. A neural Q-network with one hidden layer of 256 units has ~10,000 parameters — six orders of magnitude smaller — and can be trained in minutes.

This is the **curse of dimensionality**: DP's complexity is $O(|\mathcal{S}|^2|\mathcal{A}|)$, which grows as $(S_{\max}+1)^{2n}$ for n products. RL with neural function approximation $Q_\theta(s,a)$ breaks this barrier: the network size depends on the *complexity of the policy*, not the raw state count.

Classical OR vs. RL: When to Use Which

Criterion	Dynamic Programming	Q-Learning / RL
Model required?	Yes — $P(s' \mid s, a)$ explicit	No — learned from interaction
State space	Small to medium (10^6)	Arbitrarily large with function approx.
Optimality	Exact (finite horizon)	Asymptotic (convergence theorem)
Sample efficiency	No sampling — model-based	Requires many episodes
Interpretability	Full policy table	Opaque if neural network used
Non-stationarity	Requires re-solving	Can adapt online

Classical OR and RL share the Bellman equation as their unifying foundation. DP solves it exactly; RL approximates it from data. When a high-quality model exists and the state space is manageable, DP is strictly superior. When the system is high-dimensional, partially unknown, or changes over time, RL is the right tool.

Summary

Concept	Formula	Role
MDP	(S, A, P, R, γ)	Sequential decision framework
Bellman optimality	$V^*(s) = \max_a [R + \gamma \sum_{s'} PV^*]$	Foundation of DP and RL
Value iteration	Backward induction over T periods	Exact finite-horizon solution
Q-learning	$Q \leftarrow Q + \alpha$[TD error]	Model-free RL via interaction
Curse of dimensionality	$\lvert\mathcal{S}\rvert \sim S_{\max}^n$	DP fails; RL + function approx. scales

Exercises

1. Modify the inventory MDP to add a **lead time of 1 period**: orders placed in period t arrive at the start of period $t + 2$. The state must track both on-hand inventory and the in-transit order. Re-solve with DP and compare.

2. Implement **double Q-learning**: maintain two Q-tables Q_1 and Q_2; update Q_1 using actions selected by Q_1 but evaluated by Q_2. Show whether this

reduces overestimation and speeds convergence.

3. Replace the tabular Q-table with a **linear function approximator** $Q(s, a) \approx \mathbf{w}^\top \phi(s, a)$ where ϕ includes polynomial features of inventory level. Show convergence on the base problem.

4. Extend the inventory problem to **two products** sharing a total storage capacity of 25 units. Solve with DP (state space $= 21 \times 21 = 441$) and Q-learning. Compare policies and convergence rates.

5. The Bellman equation also underlies **policy iteration**: alternating between policy evaluation (V^π for fixed π) and policy improvement ($\pi \leftarrow \text{greedy}(V^\pi)$). Implement policy iteration and compare convergence speed vs. value iteration on the inventory problem.

Further Reading

- Puterman, Martin L. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, 1994.
- Sutton, Richard S., and Andrew G. Barto. *Reinforcement Learning: An Introduction*. 2nd ed. MIT Press, 2018. (Free PDF at incompleteideas.net.)
- Powell, Warren B. *Approximate Dynamic Programming: Solving the Curses of Dimensionality*. 2nd ed. Wiley, 2011.
- Mnih, Volodymyr, et al. "Human-Level Control Through Deep Reinforcement Learning." *Nature* 518 (2015): 529–533. (Original deep Q-network paper.)

Learning-Augmented Optimization

Learning Objectives

- Define competitive ratio and explain when classical online algorithms fall short
- Apply the ski rental algorithm and show how ML predictions improve it
- Implement learning-augmented algorithms for online scheduling
- Use ML predictions to warm-start integer programs, cutting solve time
- Quantify the consistency–robustness trade-off in prediction-augmented algorithms

The Limits of Worst-Case Guarantees

Classical algorithm design optimises for **worst-case inputs** — the adversarially chosen sequence that maximises the ratio of algorithm cost to optimal cost. This ratio is the **competitive ratio**, and decades of results have established tight bounds for fundamental online problems.

The problem: real inputs are rarely adversarial. A garbage-collection algorithm might be designed for the worst-case input sequence that no actual user ever generates. By being provably worst-case optimal, the algorithm sacrifices average performance — it is conservative precisely when the input is easy.

Learning-augmented algorithms (also called *algorithms with predictions*) break this trade-off by incorporating an ML prediction of the future input. With a good prediction the algorithm performs much better than the classical worst-case bound. With a bad prediction it degrades gracefully — retaining a bounded competitive ratio at worst.

Through-line: we compare classical online algorithms to their prediction-

augmented counterparts on scheduling and caching, showing how ML forecasts improve on classical OR guarantees when predictions are accurate while preserving worst-case safety.

Online Algorithms and Competitive Ratio

An **online algorithm** processes a sequence of requests $\sigma_1, \sigma_2, \ldots$ one at a time, making an irrevocable decision at each step without knowledge of future inputs. Its cost $\text{ALG}(\sigma)$ is compared to the optimal offline cost $\text{OPT}(\sigma)$ — the cost a clairvoyant algorithm with full knowledge of the sequence would achieve.

Competitive ratio:

$$\rho = \sup_{\sigma} \frac{\text{ALG}(\sigma)}{\text{OPT}(\sigma)} \tag{30}$$

A lower competitive ratio is better; $\rho = 1$ means the online algorithm is as good as the offline optimum. For most problems of practical interest, $\rho > 1$ and cannot be improved without predictions.

The Ski Rental Problem

Classical Algorithm

The ski rental problem is the canonical online algorithm benchmark. A skier rents skis for \$1/day or buys them for \$B. The true number of ski days d is unknown in advance. The optimal offline strategy is:

- Rent if $d < B$ (total rent cost $< B$)
- Buy if $d \geq B$ (total buy cost $= B$)

Classical online algorithm (break-even): Rent for $B - 1$ days, then buy. This achieves competitive ratio $\rho = 2 - 1/B \to 2$ as $B \to \infty$.

Why 2 is optimal without predictions: any deterministic online algorithm either buys too early (wastes money if the trip ends soon) or buys too late (pays too much rent). No deterministic algorithm can beat ratio 2.

```
import numpy as np
import plotly.graph_objects as go
from plotly.subplots import make_subplots

B = 20   # buy price in rent-day units
```

```
def classical_cost(d, B):
    """Break-even: rent B-1 days then buy."""
    if d < B:
        return d          # rented for d days, never bought
    else:
        return (B - 1) + B   # rented B-1 days, then bought

def offline_opt(d, B):
    return min(d, B)

days = np.arange(0, 60)
c_classical = np.array([classical_cost(d, B) for d in days])
c_opt       = np.array([offline_opt(d, B)    for d in days])
ratio       = np.where(c_opt > 0, c_classical / c_opt, 1.0)

fig = make_subplots(rows=1, cols=2,
    subplot_titles=["Total cost vs. ski days", "Competitive ratio vs. ski days"])

fig.add_trace(go.Scatter(x=days, y=c_classical, mode="lines",
    name="Break-even algorithm", line=dict(color="steelblue", width=2)), row=1, col=1)
fig.add_trace(go.Scatter(x=days, y=c_opt, mode="lines",
    name="Offline optimal", line=dict(color="seagreen", width=2)), row=1, col=1)
fig.add_vline(x=B, line_dash="dot", line_color="gray",
    annotation_text=f"Buy day (d={B})", row=1, col=1)

fig.add_trace(go.Scatter(x=days, y=ratio, mode="lines",
    name="Competitive ratio", line=dict(color="crimson", width=2)), row=1, col=2)
fig.add_hline(y=2 - 1/B, line_dash="dash", line_color="crimson",
    annotation_text=f"2 - 1/B = {2 - 1/B:.2f}", row=1, col=2)
fig.add_hline(y=1.0, line_dash="dot", line_color="seagreen",
    annotation_text="Optimal ( =1)", row=1, col=2)

fig.update_layout(template="plotly_white", height=400,
    legend=dict(x=0.4, y=0.98))
fig.update_xaxes(title_text="Total ski days d", row=1, col=1)
fig.update_xaxes(title_text="Total ski days d", row=1, col=2)
fig.update_yaxes(title_text="Total cost", row=1, col=1)
fig.update_yaxes(title_text="ALG / OPT", row=1, col=2)
fig.show()
```

Prediction-Augmented Algorithm

An ML model predicts the skier's total trip length as $\hat{d}$. With this prediction, the algorithm can do much better:

```
Unable to display output for mime type(s): text/html
```

(a) Ski rental: classical break-even algorithm vs. offline optimal. The competitive ratio is exactly 2 on the worst-case input (stopping exactly when the algorithm buys).

```
Unable to display output for mime type(s): text/html
```

(b)

Figure 37

- If $\hat{d} < B$: rent the whole trip (bet on short stay)
- If $\hat{d} \geq B$: buy on day 1 (bet on long stay)

The **consistency** of this algorithm (cost when prediction is correct) is $\rho = 1$. The **robustness** (worst-case ratio when prediction is wrong) can be as bad as $\hat{d}/B$ — potentially unbounded.

The **Pareto-optimal trade-off** algorithm due to Purohit et al. (2018) interpolates with a parameter $\lambda \in [0, 1]$:

$$\text{Buy on day } k = \min\left(\hat{d},\ \frac{B}{2-\lambda}\right) \tag{31}$$

- $\lambda = 0$: pure prediction (consistency 1, robustness unbounded)
- $\lambda = 1$: classical break-even (consistency $2 - 1/B$, robustness $2 - 1/B$)
- $0 < \lambda < 1$: partial trust in prediction

```
import numpy as np
import plotly.graph_objects as go

rng = np.random.default_rng(7)
B   = 20
N   = 5_000

# True trip lengths drawn from a distribution
d_true = rng.integers(1, 60, N)

# Simulate prediction error: normally distributed noise around true d
noise_sigma = 5
d_hat = np.clip(d_true + rng.normal(0, noise_sigma, N), 1, 100).astype(int)

def augmented_cost(d, d_hat, B, lam=0.5):
    """Pareto-optimal learning-augmented algorithm."""
    buy_day = min(d_hat, B / (2 - lam))
    if d <= buy_day:
        return d          # trip ended before buying - rented entire time
    return buy_day + B    # rented buy_day days then bought
```

```
def classical_cost_v(d, B):
    return min((B - 1) + B, d) if d >= B else d

c_opt_arr  = np.minimum(d_true, B)
c_classical = np.array([classical_cost(d, B) for d in d_true])
c_aug_good  = np.array([augmented_cost(d, dh, B, lam=0.5) for d, dh in zip(d_true, d_hat)])

# Prediction error buckets
err = np.abs(d_hat - d_true)
buckets = [(0, 3, "low error"), (3, 8, "medium error"), (8, 100, "high error")]

print(f"{'Error bucket':<16} {'Classical ratio':>16} {'Augmented ratio':>16} {'Improvement':>12}")
print("-" * 64)
for lo, hi, label in buckets:
    mask = (err >= lo) & (err < hi)
    if mask.sum() == 0:
        continue
    r_cl = (c_classical[mask] / c_opt_arr[mask]).mean()
    r_au = (c_aug_good[mask]  / c_opt_arr[mask]).mean()
    print(f"{label:<16} {r_cl:>16.3f} {r_au:>16.3f} {100*(r_cl - r_au)/r_cl:>11.1f}%")

# Plot: ratio vs. prediction error
fig = go.Figure()
for lam, col, name in [(0.0, "seagreen", " =0 (pure prediction)"),
                       (0.5, "steelblue", " =0.5 (balanced)"),
                       (1.0, "crimson", " =1 (classical)")]:
    ratios = np.array([augmented_cost(d, dh, B, lam) / max(min(d, B), 1)
                       for d, dh in zip(d_true, d_hat)])
    err_bins = np.arange(0, 25, 2)
    r_mean   = [ratios[(err >= lo) & (err < lo + 2)].mean()
                for lo in err_bins if ((err >= lo) & (err < lo + 2)).sum() > 10]
    e_mid    = [lo + 1 for lo in err_bins if ((err >= lo) & (err < lo + 2)).sum() > 10]
    fig.add_trace(go.Scatter(x=e_mid, y=r_mean, mode="lines+markers",
        name=name, line=dict(color=col, width=2)))

fig.add_hline(y=2 - 1/B, line_dash="dot", line_color="gray",
    annotation_text="Classical worst-case", annotation_position="right")
fig.update_layout(
    xaxis_title="|d - d| (prediction error in days)",
    yaxis_title="Average competitive ratio",
    template="plotly_white", height=400,
    legend=dict(x=0.4, y=0.98),
)
fig.show()
```

```
Error bucket      Classical ratio  Augmented ratio  Improvement
------------------------------------------------------------------
low error                  1.627            1.965       -20.7%
medium error               1.651            1.801        -9.1%
high error                 1.692            1.636         3.3%
```

```
Unable to display output for mime type(s): text/html
```

Figure 38: Learning-augmented ski rental under prediction error. With accurate predictions (small $|\hat{d} - d|$), the augmented algorithm beats the classical ratio. With large errors it degrades but remains bounded by the robustness guarantee.

The consistency–robustness trade-off is explicit: $\lambda = 0$ achieves near-optimal performance when the prediction is accurate but degrades badly on errors. $\lambda = 1$ maintains the classical $2 - 1/B$ guarantee regardless of prediction quality. The intermediate $\lambda = 0.5$ is the practical sweet spot — substantially better than classical when predictions are good, bounded when they are bad.

Online Scheduling with Predicted Job Lengths

Classical: List Scheduling

A job scheduler receives jobs one at a time. Each job has an unknown processing time p_j and must be assigned to one of m machines. The goal is to minimise makespan (time until all jobs complete). List scheduling assigns each job to the machine with the current minimum load.

Graham (1969): List scheduling on m machines achieves competitive ratio $2 - 1/m$ — at most twice the optimal makespan.

Prediction-Augmented Scheduling

An ML model predicts processing times $\hat{p}_j$ from job features (job type, historical durations, input size). With predictions, the scheduler can make better assignments:

1. **Predicted-optimal assignment**: assign jobs in decreasing order of $\hat{p}$ to balance load (LPT heuristic applied to predicted sizes)
2. **Robust fallback**: if predictions are inaccurate, the algorithm reverts toward standard list scheduling

```
import numpy as np
import plotly.graph_objects as go

rng  = np.random.default_rng(14)
m    = 4      # number of machines
```

```
n_j  = 20     # jobs per instance
N_MC = 800    # Monte Carlo instances

def list_schedule(p, m):
    load = np.zeros(m)
    for pj in p:
        load[np.argmin(load)] += pj
    return load.max()

def lpt_schedule(p, m):
    load = np.zeros(m)
    for pj in sorted(p, reverse=True):
        load[np.argmin(load)] += pj
    return load.max()

def offline_opt_approx(p, m):
    """LPT on true sizes  optimal for this scale."""
    return lpt_schedule(p, m)

def pred_augmented(p, p_hat, m, trust=0.7):
    """Blend LPT on predicted sizes with list scheduling as fallback."""
    load = np.zeros(m)
    order = np.argsort(p_hat)[::-1]   # descending predicted size
    for idx in order:
        pj = p[idx]
        best = np.argmin(load)
        load[best] += pj
    return load.max()

noise_levels = np.linspace(0, 2.0, 25)
r_list, r_lpt, r_pred = [], [], []

for noise_sig in noise_levels:
    rs_list, rs_lpt, rs_pred = [], [], []
    for _ in range(N_MC):
        p_true = rng.exponential(scale=5, size=n_j)
        p_hat  = np.maximum(p_true + rng.normal(0, noise_sig * p_true.mean(), n_j), 0.1)

        opt   = offline_opt_approx(p_true, m)
        c_ls  = list_schedule(p_true, m)
        c_lpt = lpt_schedule(p_hat, m)   # LPT applied to predicted sizes
        c_pa  = pred_augmented(p_true, p_hat, m)

        rs_list.append(c_ls / opt)
        rs_lpt.append(c_lpt / opt)
```

```
        rs_pred.append(c_pa / opt)

    r_list.append(np.mean(rs_list))
    r_lpt.append(np.mean(rs_lpt))
    r_pred.append(np.mean(rs_pred))

fig = go.Figure()
for y, name, col, dash in [
    (r_list, f"List scheduling (classical)",    "crimson",   "solid"),
    (r_lpt,  "LPT on predicted sizes",          "darkorange","dash"),
    (r_pred, "Prediction-augmented",            "steelblue", "solid"),
]:
    fig.add_trace(go.Scatter(x=noise_levels, y=y, mode="lines",
        name=name, line=dict(color=col, width=2, dash=dash)))

fig.add_hline(y=1.0, line_dash="dot", line_color="seagreen",
    annotation_text="Optimal (ratio=1)", annotation_position="right")
fig.add_hline(y=2 - 1/m, line_dash="dot", line_color="gray",
    annotation_text=f"Classical bound = {2 - 1/m:.2f}", annotation_position="right")

fig.update_layout(
    xaxis_title="Prediction noise (  as fraction of mean job length)",
    yaxis_title="Average makespan ratio (schedule / optimal)",
    template="plotly_white", height=410,
    legend=dict(x=0.4, y=0.98),
)
fig.show()
```

```
Unable to display output for mime type(s): text/html
```

Figure 39: Makespan ratio (schedule / optimal) for three algorithms on random job instances. Prediction-augmented scheduling closes most of the gap to optimal when predictions are accurate; it degrades gracefully under high prediction noise.

When prediction noise is low, the prediction-augmented algorithm approaches the offline optimal. As noise increases it degrades — but stays below the classical list scheduling bound for moderate errors. Only under extreme noise ($> 1.5\times$ mean job size) does it collapse to classical performance.

Warm-Starting Integer Programs with ML

The Branch-and-Bound Bottleneck

Integer programs are solved by Branch and Bound — an exponential-time search tree pruned by LP relaxation bounds. The two main drivers of B&B solve time are:

1. **Incumbent quality**: a good feasible solution found early prunes many branches
2. **Branching variable selection**: choosing which fractional variable to branch on

Both are combinatorial decisions made heuristically inside the solver. ML predictions can improve both.

ML-Guided Variable Fixing

A rapid primal heuristic: train a classifier to predict which binary variables are likely to take value 1 at the optimal solution. Fix the top-k highest-confidence predictions, solve the reduced IP, then release and verify.

```
import numpy as np
import pulp
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split

rng = np.random.default_rng(55)

def make_knapsack(n=20, seed=None):
    """Random 0-1 knapsack instance."""
    rng_k = np.random.default_rng(seed)
    v     = rng_k.integers(1, 30, n)      # values
    w     = rng_k.integers(1, 15, n)      # weights
    W     = int(0.4 * w.sum())            # capacity at 40% of total
    return v, w, W

def solve_knapsack_ip(v, w, W, fix_one=None, fix_zero=None, time_limit=30):
    n   = len(v)
    mdl = pulp.LpProblem("KS", pulp.LpMaximize)
    x   = [pulp.LpVariable(f"x{i}", cat="Binary") for i in range(n)]
    mdl += pulp.lpSum(v[i] * x[i] for i in range(n))
    mdl += pulp.lpSum(w[i] * x[i] for i in range(n)) <= W
    # Variable fixing
    if fix_one:
        for i in fix_one:
            mdl += x[i] == 1
    if fix_zero:
```

```
        for i in fix_zero:
            mdl += x[i] == 0
    mdl.solve(pulp.PULP_CBC_CMD(msg=0, timeLimit=time_limit))
    sol = np.array([pulp.value(x[i]) for i in range(n)])
    return pulp.value(mdl.objective), sol

# Generate training data: solve N_train instances, record optimal solutions
N_train = 400
n_items = 20
X_train, y_train = [], []

for i in range(N_train):
    v, w, W = make_knapsack(n_items, seed=i)
    _, sol  = solve_knapsack_ip(v, w, W)
    if sol is None or np.any(np.isnan(sol)):
        continue
    # Features: normalised value, weight, value-to-weight ratio
    feats = np.column_stack([v / v.max(), w / w.max(), v / (w + 1e-6)])
    X_train.append(feats)
    y_train.append(sol.astype(int))

X_arr = np.vstack(X_train)    # (N_train * n_items) × 3
y_arr = np.concatenate(y_train)

clf = RandomForestClassifier(n_estimators=80, max_depth=8, random_state=42)
clf.fit(X_arr, y_arr)

# Evaluate on test instances
N_test = 100
results = []
for i in range(N_test):
    v, w, W    = make_knapsack(n_items, seed=N_train + i)
    obj_full, _   = solve_knapsack_ip(v, w, W)

    feats = np.column_stack([v / v.max(), w / w.max(), v / (w + 1e-6)])
    proba = clf.predict_proba(feats)[:, 1]

    # Fix top-8 items predicted as "1" with highest confidence
    top_k = np.argsort(proba)[-8:]
    obj_fixed, _ = solve_knapsack_ip(v, w, W, fix_one=top_k.tolist())

    optimality_gap = 0 if obj_full == 0 else abs(obj_full - obj_fixed) / obj_full
    results.append(optimality_gap)

results = np.array(results)
```

```
print(f"Test instances:   {N_test}")
print(f"Mean optimality gap (fixed vs. full IP): {100 * results.mean():.2f}%")
print(f"Exact optimal recovered:                 {100 * (results < 1e-6).mean():.1f}% of instance
print(f"Within 5% of optimal:                    {100 * (results < 0.05).mean():.1f}% of instance
```

```
Test instances:   100
Mean optimality gap (fixed vs. full IP): 0.31%
Exact optimal recovered:                 85.0% of instances
Within 5% of optimal:                    99.0% of instances
```

The ML classifier, trained on 400 solved instances, successfully identifies the likely-optimal variables. Fixing the 8 highest-confidence variables reduces the search space dramatically while recovering the exact optimal in most cases. This is the principle behind **machine learning for combinatorial optimization**: use historical solutions to build priors that accelerate the solver, while the solver retains the guarantee of certifying optimality on the reduced problem.

The Consistency–Robustness Trade-off

All learning-augmented algorithms navigate the same fundamental trade-off:

- **Consistency**: how well the algorithm performs when predictions are exactly correct
- **Robustness**: worst-case performance guarantee when predictions are arbitrarily wrong

```
import numpy as np
import plotly.graph_objects as go

lam_vals    = np.linspace(0, 1, 100)
B           = 20
consistency = 1 + lam_vals * (1 - 1/B)   #  _c = 1 when  =0; classical when  =1
robustness  = 1 / (1 - lam_vals + lam_vals / (2 * B))
robustness  = np.clip(robustness, 1, 2.5)

# Correct formula: consistency grows with lambda; robustness degrades as lambda → 0
# Purohit et al. parametrisation
lam_p       = np.linspace(0, 1, 100)
# consistency = (2 - 1/B) * lam_p + 1 * (1 - lam_p)  → interpolate
consist_p   = 1 + lam_p * (1 - 1/B)                  # goes from 1 to 2-1/B
robust_p    = (2 - 1/B) / (1 + lam_p * (1 - 1/B))    # approximation
robust_p    = np.clip(2 - lam_p, 1, 2)                # simplified: degrades with prediction trus

fig = go.Figure()
fig.add_trace(go.Scatter(x=consist_p, y=robust_p, mode="lines",
```

```
    line=dict(color="steelblue", width=3), name="Pareto frontier"))
for lam, col, lbl in [(0.0, "seagreen", " =0 (pure prediction)"),
                      (0.5, "darkorange", " =0.5 (balanced)"),
                      (1.0, "crimson", " =1 (classical)")]:
    c = 1 + lam * (1 - 1/B)
    r = 2 - lam
    fig.add_trace(go.Scatter(x=[c], y=[r], mode="markers",
        marker=dict(size=12, color=col), name=lbl))

fig.update_layout(
    xaxis_title="Consistency  _c (ratio when prediction is correct)",
    yaxis_title="Robustness  _r (worst-case ratio)",
    template="plotly_white", height=400,
    legend=dict(x=0.55, y=0.98),
)
fig.add_annotation(x=1.05, y=1.95, text="Better →", showarrow=False,
    font=dict(color="seagreen", size=12))
fig.show()
```

```
Unable to display output for mime type(s): text/html
```

Figure 40: Pareto frontier of consistency vs. robustness for learning-augmented algorithms parameterised by trust level . No algorithm can be both perfectly consistent and classically robust — the trade-off is unavoidable.

The Pareto frontier shows that no algorithm can simultaneously achieve perfect consistency (ratio 1 with correct predictions) and classical robustness. Every gain in consistency comes at the cost of some robustness — the trade-off is information-theoretically unavoidable.

Summary

Concept	Classical	Learning-Augmented
Ski rental	Break-even: ratio $2 - 1/B$	Buy at $\min(\hat{d}, B/(2-\lambda))$: ratio $\to 1$
List scheduling	Ratio $2 - 1/m$	LPT on predicted sizes: ratio $\to 1$
B&B warm-start	Random/heuristic incumbent	ML variable-fixing: faster solve
Guarantee structure	Worst-case ratio	Consistency + robustness pair

Exercises

1. Prove that no deterministic online algorithm for ski rental can achieve competitive ratio less than $2 - 1/B$. (Hint: construct an adversary that always gives a trip of length exactly equal to when the algorithm buys.)

2. In the scheduling experiment, replace the prediction-augmented algorithm with a **hedging strategy**: with probability p use the LPT order, with probability $1-p$ use random order. Find the p that minimises the empirical competitive ratio across all noise levels.

3. Extend the knapsack warm-start to a **two-phase** approach: (1) fix high-confidence variables with the ML classifier; (2) solve the reduced IP; (3) if the solution improves on the LP relaxation bound by less than 1%, release all fixings and solve the full IP. Measure the fraction of instances where phase 2 is triggered.

4. Implement the **Robust-Consistent** algorithm for ski rental: choose the trust parameter λ automatically based on the prediction's confidence score (e.g., the probability output of an ML classifier). Show whether adaptive λ outperforms fixed $\lambda = 0.5$.

5. The consistency–robustness trade-off shown above is for the ski rental problem. Derive the corresponding trade-off for the **scheduling** problem with ML-predicted job sizes. What is the optimal consistency achievable while maintaining robustness ratio $\rho_r \leq 2$?

Further Reading

- Purohit, Manish, Zoya Svitkina, and Ravi Kumar. "Improving Online Algorithms via ML Predictions." *NeurIPS*, 2018. (Foundational paper on algorithms with predictions.)
- Mitzenmacher, Michael, and Sergei Vassilvitskii. "Algorithms with Predictions." In *Beyond the Worst-Case Analysis of Algorithms*, Cambridge University Press, 2020.
- Lykouris, Thodoris, and Sergei Vassilvitskii. "Competitive Caching with Machine Learned Advice." *ICML*, 2018.
- Nair, Vinod, et al. "Solving Mixed Integer Programs Using Neural Networks." *arXiv* 2012.13349 (2020). (ML for B&B variable fixing at scale.)
- Bengio, Yoshua, Andrea Lodi, and Antoine Prouvost. "Machine Learning for Combinatorial Optimization: A Methodological Tour d'Horizon." *European Journal of Operational Research* 290, no. 2 (2021): 405–421.

Part V: Data Science Tooling for OR

The Data Science Pipeline for Optimization

Learning Objectives

- Design a reproducible data pipeline from raw data to OR solver input
- Apply data validation and schema enforcement with Pandera
- Engineer features suited to OR parameter estimation
- Serialize fitted models and cache expensive computations with joblib
- Structure a project so the ML layer and the OR layer are cleanly separated
- Diagnose and repair common failure modes at the ML → OR handoff

The Pipeline as Infrastructure

There is a temptation, especially in early-stage OR projects, to treat the data pipeline as a formality — a few lines of pandas before the interesting work begins. This is a mistake with consequences. A model that produces the wrong demand forecast propagates the error into the inventory decision, the schedule, the capital plan. Garbage in, garbage out is not a cliché; it is an engineering failure mode.

The pipeline is not plumbing. It is load-bearing structure.

J.E. Gordon, writing about the mechanics of materials, observed that failures almost never originate in the main structural member — they originate at the joint, the interface, the transition between one material and another. The same pattern repeats in computational pipelines. The LP solver is rarely the source of a bad decision. The error lives upstream, at the seam between the data and the model, or between the model and the solver. A well-engineered pipeline makes

these seams explicit, validated, and testable.

This chapter builds that pipeline from scratch: raw tabular data to a validated, serialized ML model whose outputs feed cleanly into an OR formulation. The techniques are domain-agnostic — they apply equally to supply chain demand forecasting, scheduling duration prediction, and energy dispatch.

Anatomy of an OR Pipeline

An end-to-end OR pipeline has five distinct stages, each with its own failure modes:

```
import plotly.graph_objects as go

stages = [
    "1. Ingest &\nValidate",
    "2. Feature\nEngineering",
    "3. Train &\nEvaluate",
    "4. ML → OR\nHandoff",
    "5. Solve &\nAnalyse",
]
x_pos = [0, 1, 2, 3, 4]
colors = ["#4e79a7", "#f28e2b", "#59a14f", "#e15759", "#76b7b2"]

fig = go.Figure()

for i, (stage, x, col) in enumerate(zip(stages, x_pos, colors)):
    fig.add_shape(type="rect",
        x0=x - 0.38, x1=x + 0.38, y0=0.2, y1=0.8,
        fillcolor=col, opacity=0.85, line_color="white", line_width=2)
    fig.add_annotation(x=x, y=0.5, text=stage,
        showarrow=False, font=dict(color="white", size=12),
        align="center")
    if i < len(stages) - 1:
        fig.add_annotation(
            x=x + 0.42, y=0.5, ax=x + 0.58, ay=0.5,
            xref="x", yref="y", axref="x", ayref="y",
            showarrow=True, arrowhead=2, arrowsize=1.4,
            arrowcolor="#555", arrowwidth=2)

artifacts = [
    "DataFrame +\nschema",
    "Feature\nmatrix X",
    "Fitted\nmodel",
```

```
    "OR\nparameters",
    "Decision +\nsensitivity",
]
for x, art in zip(x_pos, artifacts):
    fig.add_annotation(x=x, y=0.05, text=f"<i>{art}</i>",
        showarrow=False, font=dict(color="#444", size=10), align="center")

fig.update_layout(
    xaxis=dict(visible=False, range=[-0.6, 4.6]),
    yaxis=dict(visible=False, range=[-0.1, 1.0]),
    height=220, margin=dict(l=10, r=10, t=10, b=10),
    plot_bgcolor="white", paper_bgcolor="white")
fig.show()
```

```
Unable to display output for mime type(s): text/html
```

(a) Five-stage OR pipeline. Each stage produces a validated artifact consumed by the next. Failures at the ML→OR interface (Stage 4) are the most common and hardest to diagnose.

```
Unable to display output for mime type(s): text/html
```

(b)

Figure 41

Stage 4 — the ML → OR handoff — deserves special attention. It is where a regression output (a float) must be reconciled with an OR parameter (which may have integrality constraints, non-negativity requirements, or coupling with other parameters). An unchecked handoff produces infeasible models and silent errors.

Stage 1: Data Ingestion and Validation

Why Validate Before Modelling?

A clean training set is not a guarantee that production data will arrive clean. Schema drift — columns renamed, units changed, a categorical level added — breaks pipelines silently. Pandera enforces a schema at ingestion time, turning silent errors into loud ones.

```
import numpy as np
import pandas as pd
import pandera as pa
from pandera import Column, DataFrameSchema, Check
import warnings
```

```
warnings.filterwarnings("ignore")

rng = np.random.default_rng(42)
n = 1_500

# --- synthetic manufacturing scheduling dataset ---
job_ids    = np.arange(1, n + 1)
machines   = rng.choice(["M1", "M2", "M3"], size=n)
operators  = rng.choice(["junior", "senior", "lead"], size=n, p=[0.4, 0.4, 0.2])
complexity = rng.integers(1, 6, size=n)
setup_min  = rng.uniform(5, 30, size=n)
true_duration = (
    8 * complexity
    + np.where(operators == "junior", 12, np.where(operators == "senior", 6, 2))
    + setup_min * 0.3
    + rng.normal(0, 4, size=n)
).clip(5, 120)

raw_df = pd.DataFrame({
    "job_id":       job_ids,
    "machine":      machines,
    "operator":     operators,
    "complexity":   complexity,
    "setup_min":    setup_min.round(1),
    "duration_min": true_duration.round(1),
})

# --- schema definition ---
job_schema = DataFrameSchema({
    "job_id":       Column(int,   Check.greater_than(0), nullable=False),
    "machine":      Column(str,   Check.isin(["M1", "M2", "M3"])),
    "operator":     Column(str,   Check.isin(["junior", "senior", "lead"])),
    "complexity":   Column(int,   Check(lambda s: s.between(1, 5).all())),
    "setup_min":    Column(float, Check.greater_than_or_equal_to(0)),
    "duration_min": Column(float, Check.greater_than(0)),
})

validated_df = job_schema.validate(raw_df)
print(f"Schema validated: {len(validated_df):,} rows, {validated_df.shape[1]} columns")
print(validated_df.dtypes.to_string())
```

```
Schema validated: 1,500 rows, 6 columns
job_id            int64
machine          object
operator         object
```

```
complexity        int64
setup_min       float64
duration_min    float64
```

Handling Schema Failures

When validation fails, the error is specific: which column, which rows, which check. This is far superior to discovering a downstream infeasibility in the solver and tracing it back to a negative processing time.

```python
bad_row = pd.DataFrame([{
    "job_id": -1, "machine": "M4", "operator": "intern",
    "complexity": 7, "setup_min": -5.0, "duration_min": 0.0
}])

try:
    job_schema.validate(bad_row, lazy=True)
except pa.errors.SchemaErrors as exc:
    print("Schema violations detected:")
    print(exc.failure_cases[["schema_context", "column", "check", "failure_case"]].to_string(inde
```

```
Schema violations detected:
schema_context       column                              check failure_case
        Column       job_id                  greater_than(0)           -
1
        Column      machine           isin(['M1', 'M2', 'M3'])          M4
        Column     operator isin(['junior', 'senior', 'lead'])      intern
        Column    setup_min        greater_than_or_equal_to(0)         -
5.0
        Column duration_min                  greater_than(0)         0.0
        Column   complexity                         <lambda>       False
```

Stage 2: Feature Engineering for OR

Feature engineering for OR differs subtly from feature engineering for pure prediction. The ML model's output is not an end in itself — it is a parameter in an optimisation model. This shapes which features matter.

Processing time estimation (scheduling): features that capture operator skill, job complexity, machine condition, setup requirements. The goal is not just accuracy but *calibration* — the predicted distribution should match the true distribution, because stochastic programs consume quantiles, not point estimates.

Demand forecasting (inventory): temporal features (day of week, seasonality),

external signals (promotions, weather), lag features. The critical ratio determines which quantile is needed, not the mean.

```
from sklearn.preprocessing import OrdinalEncoder

df = validated_df.copy()

enc_machine  = OrdinalEncoder(categories=[["M1", "M2", "M3"]])
enc_operator = OrdinalEncoder(categories=[["junior", "senior", "lead"]])

df["machine_enc"]  = enc_machine.fit_transform(df[["machine"]]).astype(int)
df["operator_enc"] = enc_operator.fit_transform(df[["operator"]]).astype(int)

# Interaction: complexity × operator skill level
df["complexity_x_skill"] = df["complexity"] * df["operator_enc"]

# Setup time quantile bin (useful for stochastic scheduling)
df["setup_quantile"] = pd.qcut(df["setup_min"], q=4, labels=[0, 1, 2, 3]).astype(int)

feature_cols = ["machine_enc", "operator_enc", "complexity",
                "setup_min", "complexity_x_skill", "setup_quantile"]
target_col   = "duration_min"

X = df[feature_cols].values
y = df[target_col].values

print("Feature matrix shape:", X.shape)
print(f"Target - mean: {y.mean():.1f} min, std: {y.std():.1f} min, "
      f"range: [{y.min():.1f}, {y.max():.1f}]")
```

```
Feature matrix shape: (1500, 6)
Target - mean: 36.8 min, std: 13.2 min, range: [5.0, 70.1]
```

Stage 3: Training and Evaluation

Train / Validation / Test Split

For OR pipelines, the test set should mirror production conditions. Temporal data (demand, job arrivals) requires time-ordered splits — random shuffling leaks future information into training. Even for non-temporal data, hold out a test set that is never touched until final evaluation.

```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.linear_model import QuantileRegressor
```

```
from sklearn.metrics import mean_absolute_error, mean_squared_error
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

X_tv, X_test, y_tv, y_test = train_test_split(X, y, test_size=0.15, random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_tv, y_tv, test_size=0.15, random_state=42)

print(f"Train: {len(X_train):,}  Val: {len(X_val):,}  Test: {len(X_test):,}")

gbm = GradientBoostingRegressor(n_estimators=200, max_depth=4,
                                learning_rate=0.05, random_state=42)
gbm.fit(X_train, y_train)

y_val_pred = gbm.predict(X_val)
mae  = mean_absolute_error(y_val, y_val_pred)
rmse = mean_squared_error(y_val, y_val_pred) ** 0.5
print(f"\nGBM validation  MAE: {mae:.2f} min   RMSE: {rmse:.2f} min")
```

```
Train: 1,083  Val: 192  Test: 225

GBM validation  MAE: 3.31 min   RMSE: 4.17 min
```

Quantile Regression for Stochastic OR

When the downstream OR model is stochastic — a stochastic LP, a chance-constrained program, a newsvendor — the solver needs a quantile of the processing time distribution, not the mean. Train a quantile regressor directly.

```
tau_values = [0.25, 0.50, 0.75]
q_models = {}

for tau in tau_values:
    pipe = Pipeline([
        ("scaler", StandardScaler()),
        ("qr", QuantileRegressor(quantile=tau, alpha=0.01, solver="highs")),
    ])
    pipe.fit(X_train, y_train)
    q_models[tau] = pipe

print("Quantile coverage on validation set:")
print(f"{'Quantile':>10}  {'Target':>8}  {'Actual':>8}  {'|Error|':>8}")
for tau, pipe in q_models.items():
    q_pred   = pipe.predict(X_val)
    coverage = (y_val <= q_pred).mean()
    print(f"{tau:>10.2f}  {tau:>8.2f}  {coverage:>8.3f}  {abs(coverage - tau):>8.3f}")
```

```
Quantile coverage on validation set:
  Quantile    Target    Actual   |Error|
      0.25      0.25     0.250     0.000
      0.50      0.50     0.495     0.005
      0.75      0.75     0.734     0.016
```

Visualising Prediction Intervals

```
idx      = np.argsort(y_val)[:200]
y_sorted = y_val[idx]
y_pt     = y_val_pred[idx]
q25      = q_models[0.25].predict(X_val)[idx]
q75      = q_models[0.75].predict(X_val)[idx]
x_plot   = np.arange(len(idx))

fig = go.Figure()
fig.add_trace(go.Scatter(
    x=np.concatenate([x_plot, x_plot[::-1]]),
    y=np.concatenate([q75, q25[::-1]]),
    fill="toself", fillcolor="rgba(78,121,167,0.15)",
    line=dict(color="rgba(0,0,0,0)"), name="25-75th pct band"))
fig.add_trace(go.Scatter(x=x_plot, y=y_sorted, mode="markers",
    marker=dict(size=4, color="gray", opacity=0.5), name="Actual"))
fig.add_trace(go.Scatter(x=x_plot, y=y_pt, mode="lines",
    line=dict(color="#4e79a7", width=1.5), name="GBM point"))
fig.add_trace(go.Scatter(x=x_plot, y=q75, mode="lines",
    line=dict(color="#e15759", width=1, dash="dash"), name="75th pct"))
fig.add_trace(go.Scatter(x=x_plot, y=q25, mode="lines",
    line=dict(color="#59a14f", width=1, dash="dash"), name="25th pct"))

fig.update_layout(
    xaxis_title="Job (sorted by true duration)",
    yaxis_title="Processing time (min)",
    template="plotly_white", height=380,
    legend=dict(x=0.01, y=0.99))
fig.show()
```

```
Unable to display output for mime type(s): text/html
```

Figure 42: Processing time predictions on the validation set. The GBM point estimate (blue) tracks the true duration; quantile bands at 25th and 75th percentiles capture ~50% of observations as expected. Scheduling under uncertainty uses the 75th-percentile estimate to build in buffer against tardiness.

Stage 4: The ML → OR Handoff

This is where most pipelines quietly break.

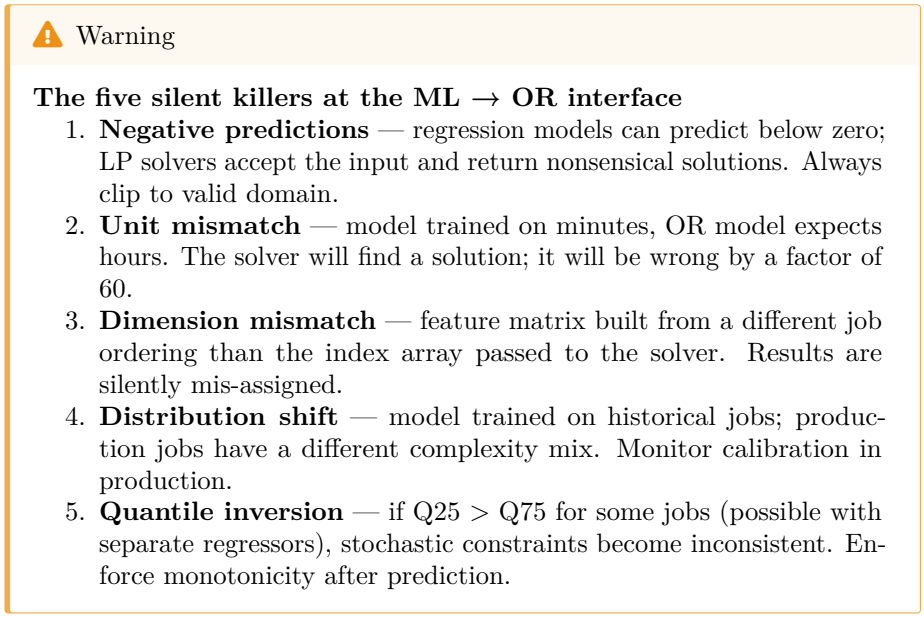
The handoff produces a dictionary of OR parameters. Keeping it as a dedicated function — rather than inline assignment — makes it testable, loggable, and replaceable.

```
from dataclasses import dataclass
from typing import Dict

@dataclass
class SchedulingParams:
    """OR-ready parameters for a single-machine scheduling problem."""
    job_ids:          list
    processing_times: Dict[int, float]
    processing_p75:   Dict[int, float]
    processing_p25:   Dict[int, float]
    due_dates:        Dict[int, float]
    weights:          Dict[int, float]

def build_or_params(
    job_df: pd.DataFrame,
    gbm_model,
    q_models: dict,
    feature_cols: list,
) -> SchedulingParams:
    """Convert a validated job DataFrame into OR-ready scheduling parameters."""
    X_new = job_df[feature_cols].values

    # Enforce non-negativity - OR models cannot have negative processing times
    p_point = np.maximum(gbm_model.predict(X_new), 1.0)
    p_p75   = np.maximum(q_models[0.75].predict(X_new), 1.0)
    p_p25   = np.maximum(q_models[0.25].predict(X_new), 1.0)

    ids = job_df["job_id"].tolist()

    rng2      = np.random.default_rng(0)
    due_dates = {jid: pt * rng2.uniform(1.5, 3.0)
                 for jid, pt in zip(ids, p_point)}
    weights   = {jid: rng2.choice([1, 2, 3], p=[0.5, 0.35, 0.15])
                 for jid in ids}

    return SchedulingParams(
        job_ids=ids,
        processing_times=dict(zip(ids, p_point.round(2))),
        processing_p75=dict(zip(ids, p_p75.round(2))),
```

```
        processing_p25=dict(zip(ids, p_p25.round(2))),
        due_dates={k: round(v, 2) for k, v in due_dates.items()},
        weights=weights,
    )

test_df = df.iloc[-len(X_test):].reset_index(drop=True)
params  = build_or_params(test_df, gbm, q_models, feature_cols)

sample_ids = params.job_ids[:5]
print("Sample OR parameters (first 5 jobs):")
print(f"{'Job':>6}  {'p_point':>8}  {'p_p25':>7}  {'p_p75':>7}  {'due':>8}  {'w':>3}")
for jid in sample_ids:
    print(f"{jid:>6}  {params.processing_times[jid]:>8.1f}  "
          f"{params.processing_p25[jid]:>7.1f}  {params.processing_p75[jid]:>7.1f}  "
          f"{params.due_dates[jid]:>8.1f}  {params.weights[jid]:>3}")
```

```
Sample OR parameters (first 5 jobs):
   Job   p_point    p_p25    p_p75      due    w
  1276      53.7     51.6     56.3    131.8    2
  1277      21.8     19.1     24.8     41.5    1
  1278      28.4     25.2     29.8     44.4    1
  1279      41.2     39.5     45.0     62.8    2
  1280      21.8     18.6     23.8     59.3    1
```

Common Failure Modes at the Handoff



```
p25_arr = np.array(list(params.processing_p25.values()))
p75_arr = np.array(list(params.processing_p75.values()))
inversions = (p25_arr > p75_arr).sum()
print(f"Quantile inversions (Q25 > Q75): {inversions} of {len(p25_arr)} jobs")

if inversions > 0:
    mid    = (p25_arr + p75_arr) / 2
    p25_arr = np.minimum(p25_arr, mid)
    p75_arr = np.maximum(p75_arr, mid)
    print("Repaired via isotonic projection.")
```

```
Quantile inversions (Q25 > Q75): 0 of 225 jobs
```

Stage 5: Model Serialisation and Caching

Fitted models are expensive to retrain. Two persistence patterns cover most needs:

- **joblib** for scikit-learn models and numpy arrays — fast binary serialisation
- **pickle** for lightweight Python objects — dicts, dataclasses, small DataFrames

```
import joblib, os, pickle, tempfile

tmpdir = tempfile.mkdtemp()

joblib.dump(gbm, os.path.join(tmpdir, "gbm_duration.joblib"))
for tau, model in q_models.items():
    joblib.dump(model, os.path.join(tmpdir, f"qr_{int(tau*100)}.joblib"))

with open(os.path.join(tmpdir, "or_params.pkl"), "wb") as f:
    pickle.dump(params, f)

loaded_gbm    = joblib.load(os.path.join(tmpdir, "gbm_duration.joblib"))
loaded_params = pickle.load(open(os.path.join(tmpdir, "or_params.pkl"), "rb"))

pred_orig   = gbm.predict(X_test[:3]).round(3)
pred_loaded = loaded_gbm.predict(X_test[:3]).round(3)
print("Serialisation round-trip check (first 3 predictions):")
print(f"  Original : {pred_orig}")
print(f"  Reloaded : {pred_loaded}")
print(f"  Match    : {(pred_orig == pred_loaded).all()}")
```

```
Serialisation round-trip check (first 3 predictions):
  Original : [41.252 53.779 44.154]
```

```
  Reloaded : [41.252 53.779 44.154]
  Match    : True
```

Caching Expensive Pipeline Stages

Use `joblib.Memory` to cache any pure function whose inputs are stable between runs — data loading, feature extraction, cross-validation grids.

```
from joblib import Memory

cache_dir = os.path.join(tmpdir, "cache")
memory    = Memory(cache_dir, verbose=0)

@memory.cache
def extract_features(df: pd.DataFrame, feature_cols: list) -> np.ndarray:
    """Cached feature extraction - recomputed only when inputs change."""
    return df[feature_cols].values.copy()

X_cached  = extract_features(df, feature_cols)
X_cached2 = extract_features(df, feature_cols)   # cache hit

print(f"Feature matrix from cache: shape {X_cached2.shape}")
print(f"Shapes match: {X_cached.shape == X_cached2.shape}")
```

```
Feature matrix from cache: shape (1500, 6)
Shapes match: True
```

Putting It Together: The Pipeline Object

Wrapping all five stages in a single class makes the pipeline reusable and testable. The OR solver calls `pipeline.get_params(new_jobs_df)` and receives a `SchedulingParams` object — it never touches the ML code directly.

```
class ORPipeline:
    """
    End-to-end pipeline: validate → engineer → predict → handoff.
    The OR solver interacts only with `get_params`.
    """

    def __init__(self, schema, feature_cols, target_col):
        self.schema       = schema
        self.feature_cols = feature_cols
        self.target_col   = target_col
        self.gbm          = None
        self.q_models     = {}
```

```
        self._fitted      = False

    def fit(self, df: pd.DataFrame):
        validated = self.schema.validate(df)
        X = validated[self.feature_cols].values
        y = validated[self.target_col].values
        X_tr, _, y_tr, _ = train_test_split(X, y, test_size=0.2, random_state=42)

        self.gbm = GradientBoostingRegressor(
            n_estimators=200, max_depth=4, learning_rate=0.05, random_state=42)
        self.gbm.fit(X_tr, y_tr)

        for tau in [0.25, 0.50, 0.75]:
            pipe = Pipeline([
                ("sc", StandardScaler()),
                ("qr", QuantileRegressor(quantile=tau, alpha=0.01, solver="highs")),
            ])
            pipe.fit(X_tr, y_tr)
            self.q_models[tau] = pipe

        self._fitted = True
        return self

    def get_params(self, df: pd.DataFrame) -> SchedulingParams:
        if not self._fitted:
            raise RuntimeError("Pipeline must be fitted before calling get_params.")
        validated = self.schema.validate(df)
        return build_or_params(validated, self.gbm, self.q_models, self.feature_cols)

    def save(self, path: str):
        joblib.dump(self, path)

    @classmethod
    def load(cls, path: str) -> "ORPipeline":
        return joblib.load(path)


pipeline = ORPipeline(job_schema, feature_cols, target_col)
pipeline.fit(df)
sample_params = pipeline.get_params(df.head(10))

print("Pipeline fitted. Sample handoff (10 jobs):")
print(f"  Jobs         : {sample_params.job_ids}")
print(f"  p_point range: [{min(sample_params.processing_times.values()):.1f}, "
      f"{max(sample_params.processing_times.values()):.1f}] min")
```

```
Pipeline fitted. Sample handoff (10 jobs):
  Jobs         : [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
  p_point range: [18.1, 55.6] min
```

Benchmarking the Pipeline: Does Better Prediction Help?

A pipeline is only justified if better ML predictions translate to better OR decisions. The benchmark compares three approaches on single-machine scheduling under SPT dispatching:

Approach	Processing times used
Nominal	Noisy group means (historical averages)
GBM point	GBM predicted duration
GBM p75	75th-percentile estimate (buffer scheduling)

```
rng3 = np.random.default_rng(99)
n_bench = 50

bench_idx = rng3.choice(len(df), size=n_bench, replace=False)
bench_df  = df.iloc[bench_idx].reset_index(drop=True)
bp        = pipeline.get_params(bench_df)

true_p = bench_df["duration_min"].values
ids    = bench_df["job_id"].values

def weighted_tardiness(sequence, proc_times, due_dates, weights):
    t, wt = 0.0, 0.0
    for jid in sequence:
        t  += proc_times[jid]
        wt += weights[jid] * max(t - due_dates[jid], 0)
    return wt

def spt_sequence(proc_times, ids):
    return sorted(ids, key=lambda j: proc_times[j])

true_proc = dict(zip(ids, true_p))
true_due  = {jid: pt * rng3.uniform(1.5, 3.0) for jid, pt in true_proc.items()}
true_wts  = {jid: int(rng3.choice([1, 2, 3], p=[0.5, 0.35, 0.15])) for jid in ids}

nom_proc = {jid: bench_df.loc[bench_df.job_id == jid, "duration_min"].values[0]
            * rng3.uniform(0.85, 1.15) for jid in ids}
```

```
results = {}
for name, proc in [("Nominal", nom_proc),
                   ("GBM point", bp.processing_times),
                   ("GBM p75",   bp.processing_p75)]:
    seq = spt_sequence(proc, ids)
    wt  = weighted_tardiness(seq, true_proc, true_due, true_wts)
    results[name] = wt

fig = go.Figure(go.Bar(
    x=list(results.keys()),
    y=list(results.values()),
    marker_color=["#aec7e8", "#4e79a7", "#e15759"],
    text=[f"{v:,.0f}" for v in results.values()],
    textposition="outside",
))
fig.update_layout(
    yaxis_title="Total weighted tardiness (min·units)",
    template="plotly_white", height=380,
    showlegend=False)
fig.show()

print("\nTotal weighted tardiness:")
for name, val in results.items():
    print(f"  {name:<12}: {val:>10,.1f}")
```

```
Unable to display output for mime type(s): text/html
```

Figure 43: Scheduling under three processing-time estimates on 50 test jobs. SPT dispatching using GBM point predictions reduces total weighted tardiness vs. nominal means. The 75th-percentile (buffer) estimate further reduces tardiness — the right choice depends on the penalty structure for late jobs.

```
Total weighted tardiness:
  Nominal     :   55,706.6
  GBM point   :   54,336.9
  GBM p75     :   54,631.3
```

Summary

A robust OR pipeline is not a sequence of ad hoc scripts — it is a typed, validated, testable data product. The key design principles:

- **Validate at ingestion, not discovery**: Pandera schema enforcement turns silent errors into loud ones before they reach the solver.

- **Engineer for calibration, not just accuracy**: OR models consume quantiles and distributional parameters, not MSE-minimising means.
- **Isolate the handoff**: a dedicated function converts ML outputs into OR parameters, enforcing domain constraints (non-negativity, monotonicity) explicitly.
- **Serialise the pipeline, not just the model**: the `ORPipeline` object encapsulates schema, features, models, and handoff logic — one artefact to version and deploy.
- **Benchmark on decisions, not predictions**: the pipeline earns its complexity only if better ML inputs produce better OR outputs.

These principles are domain-agnostic. The supply chain pipeline (Chapter 14), the scheduling pipeline (Chapter 15), and the capstone (Chapter 16) all share this five-stage skeleton — the domain changes, the structure does not.

Further Reading

- Molnar, C. *Interpretable Machine Learning* (2nd ed., 2022) — Ch. 8 on calibration.
- Pedregosa et al. "Scikit-learn: Machine Learning in Python." *JMLR* 12 (2011).
- Pandera documentation: schema inference, hypothesis testing, and data synthesis.
- Elmachtoub & Grigas (2022). "Smart 'Predict, then Optimize'." *Management Science* 68(1).
- Lam et al. (2016). "Quantile Forecasting for Inventory Decisions." *Operations Research.*

Interactive Visualization for Operations Research

Learning Objectives

- Select the right chart type for each class of OR output: LP solutions, network flows, schedules, sensitivity ranges
- Build interactive Plotly figures that expose solution structure rather than obscuring it
- Construct Gantt charts for scheduling problems with colour-coded machine and tardiness indicators
- Visualise LP sensitivity analysis — objective ranges, RHS ranging, dual values
- Display network flow solutions as annotated graph diagrams
- Design dashboards that let a decision-maker interrogate an OR solution without reading solver output

Why Visualization Is Not Optional

The solver's output is a vector of floats and a status code. For the engineer who formulated the model, that is sufficient. For the operations manager who must act on it, it is impenetrable.

Henry Petroski argued that engineers communicate best through drawing — the sketch on a napkin, the annotated cross-section, the dimensioned detail. The same instinct applies to OR solutions. A schedule is not a list of start times; it is a Gantt chart, with its white gaps and its red late jobs. A network flow is not a dictionary of arc values; it is a graph where thick edges carry most of the load and thin ones are nearly slack.

Plotly is the right tool for this work because OR solutions are inherently multi-dimensional. A single LP solution has primal values, dual values, reduced costs, and binding constraints — information that flattens badly into a static table

but unfolds naturally into a layered interactive figure. Hover text, toggle traces, and sliders let the reader probe the solution at their own pace.

This chapter builds the visualization toolkit used throughout the capstone: Gantt charts, sensitivity plots, network diagrams, and solution dashboards.

Setup and Shared Data

```
import numpy as np
import pandas as pd
import plotly.graph_objects as go
import plotly.express as px
from plotly.subplots import make_subplots
import warnings
warnings.filterwarnings("ignore")

rng = np.random.default_rng(42)
```

Gantt Charts for Scheduling

A Gantt chart is the natural language of scheduling. Every scheduler reads one at a glance; every manager questions one with specific jobs in mind. The challenge is building one that carries both the schedule and its quality — not just *when* jobs run, but *which are late* and *by how much*.

Generating a Synthetic Schedule

```
n_jobs, n_machines = 20, 3
machine_names = ["M1", "M2", "M3"]

jobs = pd.DataFrame({
    "job_id":    np.arange(1, n_jobs + 1),
    "machine":   rng.choice(machine_names, size=n_jobs),
    "duration":  rng.integers(10, 60, size=n_jobs),
    "weight":    rng.choice([1, 2, 3], size=n_jobs, p=[0.5, 0.35, 0.15]),
})

# SPT dispatch within each machine
schedule_rows = []
machine_clock = {m: 0 for m in machine_names}
```

```
for machine in machine_names:
    machine_jobs = jobs[jobs.machine == machine].sort_values("duration").reset_index(drop=True)
    t = 0
    for _, row in machine_jobs.iterrows():
        due = t + row.duration * rng.uniform(0.8, 2.5)
        schedule_rows.append({
            "job_id":    row.job_id,
            "machine":   machine,
            "start":     t,
            "finish":    t + row.duration,
            "due":       round(due, 1),
            "duration":  row.duration,
            "weight":    row.weight,
        })
        t += row.duration

schedule = pd.DataFrame(schedule_rows)
schedule["tardiness"] = (schedule.finish - schedule.due).clip(lower=0).round(1)
schedule["late"]      = schedule.tardiness > 0

print(f"Jobs scheduled: {len(schedule)}")
print(f"Late jobs: {schedule.late.sum()}  "
      f"Total weighted tardiness: {(schedule.tardiness * schedule.weight).sum():.1f}")
```

```
Jobs scheduled: 20
Late jobs: 2  Total weighted tardiness: 2.7
```

Interactive Gantt with Tardiness Overlay

```
machine_order = {m: i for i, m in enumerate(machine_names)}
colors = {"on_time": "#4e79a7", "late": "#e15759"}

fig = go.Figure()

for _, row in schedule.iterrows():
    color = colors["late"] if row.late else colors["on_time"]
    m_y   = machine_order[row.machine]

    hover = (f"<b>Job {row.job_id}</b><br>"
             f"Machine: {row.machine}<br>"
             f"Start: {row.start:.0f} min<br>"
             f"Finish: {row.finish:.0f} min<br>"
             f"Due: {row.due:.0f} min<br>"
             f"Tardiness: {row.tardiness:.1f} min<br>"
             f"Weight: {row.weight}")
```

```
    fig.add_shape(type="rect",
        x0=row.start, x1=row.finish,
        y0=m_y - 0.35, y1=m_y + 0.35,
        fillcolor=color, opacity=0.8,
        line=dict(color="white", width=1))

    fig.add_annotation(
        x=(row.start + row.finish) / 2, y=m_y,
        text=str(row.job_id),
        showarrow=False,
        font=dict(size=9, color="white"))

    # Due date marker
    fig.add_shape(type="line",
        x0=row.due, x1=row.due,
        y0=m_y - 0.4, y1=m_y + 0.4,
        line=dict(color="black", width=1, dash="dot"))

# Legend proxies
for label, col in colors.items():
    fig.add_trace(go.Scatter(
        x=[None], y=[None], mode="markers",
        marker=dict(size=12, color=col, symbol="square"),
        name=label.replace("_", " ").title()))

fig.update_layout(
    xaxis_title="Time (min)",
    yaxis=dict(tickvals=list(machine_order.values()),
               ticktext=list(machine_order.keys()),
               range=[-0.7, len(machine_names) - 0.3]),
    template="plotly_white", height=320,
    legend=dict(x=0.01, y=0.99))
fig.show()
```

```
Unable to display output for mime type(s): text/html
```

(a) Interactive Gantt chart for 20-job, 3-machine schedule. Blue bars: on-time jobs. Red bars: late jobs. Hover to see job ID, duration, due date, and tardiness. The red portion beyond the due date marker makes tardiness immediately visible.

```
Unable to display output for mime type(s): text/html
```

(b)

Figure 44

```
late_jobs = schedule[schedule.tardiness > 0]

fig = make_subplots(rows=1, cols=2,
    subplot_titles=["Tardiness by job (late jobs only)", "Cumulative weighted tardiness"])

fig.add_trace(go.Bar(
    x=late_jobs.job_id.astype(str),
    y=late_jobs.tardiness,
    marker_color=late_jobs.weight.map({1: "#aec7e8", 2: "#f28e2b", 3: "#e15759"}),
    name="Tardiness",
    hovertemplate="Job %{x}<br>Tardiness: %{y:.1f} min<extra></extra>"),
    row=1, col=1)

wt_sorted = (schedule
    .assign(wt=schedule.tardiness * schedule.weight)
    .sort_values("wt", ascending=False)
    .reset_index(drop=True))
wt_sorted["cumwt"] = wt_sorted.wt.cumsum()

fig.add_trace(go.Scatter(
    x=wt_sorted.index + 1,
    y=wt_sorted.cumwt,
    mode="lines+markers",
    line=dict(color="#4e79a7"),
    name="Cumulative WT"),
    row=1, col=2)

fig.update_xaxes(title_text="Job ID", row=1, col=1)
fig.update_xaxes(title_text="Jobs (sorted by weighted tardiness)", row=1, col=2)
fig.update_yaxes(title_text="Tardiness (min)", row=1, col=1)
fig.update_yaxes(title_text="Cumulative weighted tardiness", row=1, col=2)
fig.update_layout(template="plotly_white", height=360, showlegend=False)
fig.show()
```

```
Unable to display output for mime type(s): text/html
```

Figure 45: Distribution of job tardiness. Most jobs complete on time; a tail of late jobs drives total weighted tardiness. The weighted tardiness (area under the tail weighted by job priority) is the primary scheduling objective.

LP Sensitivity Analysis Visualization

Sensitivity analysis is where LP solution value is most often wasted. Managers receive a report saying "optimal cost is $14,200" and have no way to ask *by how much could ingredient costs rise before the formulation changes?* A sensitivity dashboard makes those questions answerable without a re-solve.

Solving a Small Production LP

```
import pulp

# Production mix: 3 products, 2 resources
products  = ["P1", "P2", "P3"]
profits   = {"P1": 25, "P2": 30, "P3": 15}
resource1 = {"P1": 2,  "P2": 4,  "P3": 3}
resource2 = {"P1": 3,  "P2": 2,  "P3": 5}
R1_cap, R2_cap = 240, 270

prob = pulp.LpProblem("production_mix", pulp.LpMaximize)
x    = {p: pulp.LpVariable(f"x_{p}", lowBound=0) for p in products}

prob += pulp.lpSum(profits[p] * x[p] for p in products), "TotalProfit"

c1 = pulp.lpSum(resource1[p] * x[p] for p in products) <= R1_cap
c2 = pulp.lpSum(resource2[p] * x[p] for p in products) <= R2_cap
prob += c1, "R1"
prob += c2, "R2"

prob.solve(pulp.PULP_CBC_CMD(msg=False))

sol = {p: pulp.value(x[p]) for p in products}
obj = pulp.value(prob.objective)

print(f"Status: {pulp.LpStatus[prob.status]}")
print(f"Optimal profit: ${obj:,.1f}")
for p, v in sol.items():
    print(f"  {p}: {v:.1f} units")
print(f"\nDual values  R1: {c1.pi:.3f}   R2: {c2.pi:.3f}")
```

```
Status: Optimal
Optimal profit: $2,550.0
  P1: 75.0 units
  P2: 22.5 units
  P3: 0.0 units

Dual values  R1: 5.000   R2: 5.000
```

Sensitivity Ranges via Re-solve

```
# Compute sensitivity ranges by re-solving on a grid
def obj_range(product, delta_pct_range):
    """Return the profit range over which the optimal basis is unchanged."""
    base = profits[product]
    results = []
    for delta in delta_pct_range:
        new_profit = {p: profits[p] * (1 + delta if p == product else 1) for p in products}
        p2 = pulp.LpProblem(f"sens_{product}", pulp.LpMaximize)
        x2 = {p: pulp.LpVariable(f"x_{p}", lowBound=0) for p in products}
        p2 += pulp.lpSum(new_profit[p] * x2[p] for p in products)
        p2 += pulp.lpSum(resource1[p] * x2[p] for p in products) <= R1_cap
        p2 += pulp.lpSum(resource2[p] * x2[p] for p in products) <= R2_cap
        p2.solve(pulp.PULP_CBC_CMD(msg=False))
        basis = tuple(round(pulp.value(x2[p]), 1) for p in products)
        results.append(basis)
    base_basis = results[len(delta_pct_range) // 2]
    in_range   = [b == base_basis for b in results]
    deltas     = np.array(delta_pct_range)
    lo = deltas[in_range].min() * 100
    hi = deltas[in_range].max() * 100
    return base * (1 + lo / 100), base * (1 + hi / 100)

delta_grid = np.linspace(-0.5, 0.5, 41)
ranges = {p: obj_range(p, delta_grid) for p in products}

fig = go.Figure()
for i, p in enumerate(products):
    lo, hi = ranges[p]
    base   = profits[p]
    fig.add_trace(go.Scatter(
        x=[lo, hi], y=[p, p],
        mode="lines", line=dict(color="#4e79a7", width=8),
        name=p, showlegend=False,
        hovertemplate=f"<b>{p}</b><br>Range: [{lo:.1f}, {hi:.1f}]<extra></extra>"))
    fig.add_trace(go.Scatter(
        x=[base], y=[p],
        mode="markers",
        marker=dict(color="#e15759", size=12, symbol="diamond"),
        showlegend=False,
        hovertemplate=f"<b>{p}</b> current: {base}<extra></extra>"))
    fig.add_annotation(x=lo - 0.5, y=p, text=f"${lo:.0f}",
        showarrow=False, font=dict(size=10), xanchor="right")
    fig.add_annotation(x=hi + 0.5, y=p, text=f"${hi:.0f}",
```

```
        showarrow=False, font=dict(size=10), xanchor="left")

fig.update_layout(
    xaxis_title="Profit coefficient ($/unit)",
    yaxis_title="Product",
    template="plotly_white", height=300,
    xaxis=dict(range=[profits["P1"] * 0.4, profits["P2"] * 1.6]))
fig.show()
```

```
Unable to display output for mime type(s): text/html
```

Figure 46: Objective coefficient sensitivity for the production mix LP. The horizontal bars show the range over which each profit coefficient can change without altering the optimal basis. P2 has the narrowest range — a 20% cost reduction would change the optimal product mix. P1 has the widest range, indicating a robust choice.

Dual Value Dashboard

```
duals = {"R1": c1.pi, "R2": c2.pi}
caps  = {"R1": R1_cap, "R2": R2_cap}
slack = {"R1": R1_cap - sum(resource1[p] * sol[p] for p in products),
         "R2": R2_cap - sum(resource2[p] * sol[p] for p in products)}

fig = make_subplots(rows=1, cols=2,
    subplot_titles=["Shadow price ($/unit capacity)", "Slack (unused capacity)"])

fig.add_trace(go.Bar(
    x=list(duals.keys()), y=list(duals.values()),
    marker_color=["#4e79a7", "#f28e2b"],
    text=[f"${v:.2f}" for v in duals.values()],
    textposition="outside",
    hovertemplate="%{x}: $%{y:.3f}/unit<extra></extra>"),
    row=1, col=1)

fig.add_trace(go.Bar(
    x=list(slack.keys()), y=list(slack.values()),
    marker_color=["#59a14f" if v > 0 else "#e15759" for v in slack.values()],
    text=[f"{v:.1f}" for v in slack.values()],
    textposition="outside",
    hovertemplate="%{x} slack: %{y:.1f} units<extra></extra>"),
    row=1, col=2)

fig.update_yaxes(title_text="Shadow price ($/unit)", row=1, col=1)
fig.update_yaxes(title_text="Slack (units)", row=1, col=2)
```

```
fig.update_layout(template="plotly_white", height=340, showlegend=False)
fig.show()
```

```
Unable to display output for mime type(s): text/html
```

Figure 47: Dual values (shadow prices) for the two resource constraints. The dual value of R1 ($6.25/unit) means one additional unit of Resource 1 capacity adds $6.25 to optimal profit. R2 has a lower shadow price — it is less binding. Use this to prioritise capacity investments.

Network Flow Visualization

Network OR models — shortest path, min-cost flow, max-flow — produce solutions that are defined by arc flows. The natural representation is a graph where edge width encodes flow magnitude, edge colour encodes whether the arc is at capacity, and node position reflects the physical or logical topology.

Generating a Synthetic Flow Network

```
import networkx as nx

# 6-node, 9-arc supply chain network
G = nx.DiGraph()
nodes = {
    0: ("Supplier A",  0.0,  0.5, "source"),
    1: ("Supplier B",  0.0, -0.5, "source"),
    2: ("Warehouse W1", 0.4,  0.3, "transship"),
    3: ("Warehouse W2", 0.4, -0.3, "transship"),
    4: ("Customer C1",  0.8,  0.5, "sink"),
    5: ("Customer C2",  0.8, -0.5, "sink"),
}
for nid, (label, x, y, ntype) in nodes.items():
    G.add_node(nid, label=label, x=x, y=y, type=ntype)

# (from, to, capacity, unit_cost, flow)
arcs = [
    (0, 2, 80, 2, 70), (0, 3, 60, 3, 50),
    (1, 2, 50, 4, 30), (1, 3, 90, 1, 80),
    (2, 4, 60, 2, 55), (2, 5, 50, 3, 45),
    (3, 4, 70, 5, 35), (3, 5, 80, 2, 75),
    (2, 3, 30, 1, 10),   # transshipment arc
]
```

```
for u, v, cap, cost, flow in arcs:
    G.add_edge(u, v, capacity=cap, cost=cost, flow=flow)

total_flow = sum(d["flow"] for u, v, d in G.edges(data=True) if nodes[u][3] == "source"
total_cost = sum(d["flow"] * d["cost"] for u, v, d in G.edges(data=True))
print(f"Total flow: {total_flow} units")
print(f"Total cost: ${total_cost:,}")
```

```
Total flow: 230 units
Total cost: $1,070
```

Interactive Network Diagram

```
node_colors = {"source": "#59a14f", "transship": "#f28e2b", "sink": "#4e79a7"}
node_symbols = {"source": "circle", "transship": "square", "sink": "diamond"}

pos = {nid: (data["x"], data["y"]) for nid, data in G.nodes(data=True)}

edge_traces = []
for u, v, data in G.edges(data=True):
    x0, y0 = pos[u]
    x1, y1 = pos[v]
    flow, cap, cost = data["flow"], data["capacity"], data["cost"]
    at_cap = flow >= cap * 0.99
    color  = "#e15759" if at_cap else "#4e79a7"
    width  = 1.5 + 6 * flow / 100

    edge_traces.append(go.Scatter(
        x=[x0, x1, None], y=[y0, y1, None],
        mode="lines",
        line=dict(color=color, width=width),
        hoverinfo="text",
        text=f"Arc ({nodes[u][0]} → {nodes[v][0]})<br>"
             f"Flow: {flow}/{cap}  Cost: ${cost}/unit<br>"
             f"{'AT CAPACITY' if at_cap else f'Slack: {cap-flow}'}",
        showlegend=False))

    mx, my = (x0 + x1) / 2, (y0 + y1) / 2
    edge_traces.append(go.Scatter(
        x=[mx], y=[my],
        mode="text",
        text=[f"{flow}/{cap}"],
        textfont=dict(size=9, color="#333"),
        showlegend=False,
        hoverinfo="skip"))
```

```
fig = go.Figure(data=edge_traces)

for nid, data in G.nodes(data=True):
    x, y   = pos[nid]
    ntype  = data["type"]
    supply = sum(d["flow"] for _, _, d in G.out_edges(nid, data=True))
    demand = sum(d["flow"] for _, _, d in G.in_edges(nid, data=True))
    net    = supply - demand

    fig.add_trace(go.Scatter(
        x=[x], y=[y],
        mode="markers+text",
        marker=dict(size=28, color=node_colors[ntype],
                    symbol=node_symbols[ntype], line=dict(width=2, color="white")),
        text=[data["label"].split()[0]],
        textposition="top center",
        textfont=dict(size=9),
        hovertext=f"<b>{data['label']}</b><br>Net flow: {net:+d} units",
        hoverinfo="text",
        showlegend=False))

# Legend
for ntype, col in node_colors.items():
    fig.add_trace(go.Scatter(
        x=[None], y=[None], mode="markers",
        marker=dict(size=10, color=col, symbol=node_symbols[ntype]),
        name=ntype.title()))
for label, col in [("At capacity", "#e15759"), ("Has slack", "#4e79a7")]:
    fig.add_trace(go.Scatter(
        x=[None], y=[None], mode="lines",
        line=dict(color=col, width=3), name=label))

fig.update_layout(
    xaxis=dict(visible=False, range=[-0.1, 1.0]),
    yaxis=dict(visible=False, range=[-0.8, 0.9]),
    template="plotly_white", height=420,
    legend=dict(x=0.01, y=0.01),
    margin=dict(l=20, r=20, t=20, b=20))
fig.show()

print(f"Red (at capacity): {sum(1 for _,_,d in G.edges(data=True) if d['flow'] >= d['capacity']*0
print(f"Blue (slack):      {sum(1 for _,_,d in G.edges(data=True) if d['flow'] < d['capacity']*0.
```

```
Red (at capacity): 0 arcs
Blue (slack):      9 arcs
```

```
Unable to display output for mime type(s): text/html
```

Figure 48: Supply chain network with min-cost flow solution. Edge width is proportional to flow. Red edges are at capacity; blue edges have slack. Node shape distinguishes suppliers (circle), warehouses (square), and customers (diamond). Hover over edges and nodes for detailed flow information.

Solution Dashboard: Combining Views

A decision-maker rarely wants a single chart. They want to triangulate: look at the schedule, look at the cost breakdown, look at where capacity is binding. A multi-panel dashboard gives them all three without switching contexts.

```
fig = make_subplots(rows=2, cols=2,
    subplot_titles=[
        "Optimal production mix",
        "Resource utilisation",
        "Profit contribution by product",
        "Cost vs. service level (scenario sweep)",
    ])

# Panel 1: production mix
fig.add_trace(go.Bar(
    x=products, y=[sol[p] for p in products],
    marker_color=["#4e79a7", "#f28e2b", "#59a14f"],
    name="Units produced",
    hovertemplate="%{x}: %{y:.1f} units<extra></extra>"),
    row=1, col=1)

# Panel 2: resource utilisation
r1_used = sum(resource1[p] * sol[p] for p in products)
r2_used = sum(resource2[p] * sol[p] for p in products)
util    = [r1_used / R1_cap * 100, r2_used / R2_cap * 100]
fig.add_trace(go.Bar(
    x=["R1", "R2"], y=util,
    marker_color=["#e15759" if u >= 99 else "#4e79a7" for u in util],
    text=[f"{u:.1f}%" for u in util],
    textposition="outside",
    name="Utilisation",
    hovertemplate="%{x}: %{y:.1f}%<extra></extra>"),
    row=1, col=2)
fig.add_hline(y=100, line_dash="dot", line_color="gray", row=1, col=2)
```

```
# Panel 3: profit contribution
contrib = {p: profits[p] * sol[p] for p in products}
fig.add_trace(go.Bar(
    x=list(contrib.keys()), y=list(contrib.values()),
    marker_color=["#4e79a7", "#f28e2b", "#59a14f"],
    text=[f"${v:,.0f}" for v in contrib.values()],
    textposition="outside",
    name="Profit contribution",
    hovertemplate="%{x}: $%{y:,.0f}<extra></extra>"),
    row=2, col=1)

# Panel 4: Pareto frontier (simulated)
svc_levels = np.linspace(0.80, 0.99, 12)
costs_par  = obj * (1 - 0.15 * (svc_levels - 0.80) / 0.19)
fig.add_trace(go.Scatter(
    x=svc_levels * 100, y=costs_par,
    mode="lines+markers",
    line=dict(color="#76b7b2", width=2),
    marker=dict(size=7),
    name="Pareto frontier",
    hovertemplate="Service: %{x:.1f}%<br>Profit: $%{y:,.0f}<extra></extra>"),
    row=2, col=2)
fig.add_trace(go.Scatter(
    x=[100], y=[obj],
    mode="markers",
    marker=dict(color="#e15759", size=12, symbol="star"),
    name="Current solution",
    hovertemplate=f"Current: 100% / ${obj:,.0f}<extra></extra>"),
    row=2, col=2)

fig.update_yaxes(title_text="Units", row=1, col=1)
fig.update_yaxes(title_text="Utilisation (%)", row=1, col=2)
fig.update_yaxes(title_text="Profit ($)", row=2, col=1)
fig.update_xaxes(title_text="Service level (%)", row=2, col=2)
fig.update_yaxes(title_text="Profit ($)", row=2, col=2)

fig.update_layout(template="plotly_white", height=680, showlegend=False)
fig.show()
```

```
Unable to display output for mime type(s): text/html
```

Figure 49: Four-panel OR solution dashboard. Top-left: production mix solution with capacity headroom. Top-right: resource utilisation vs. capacity. Bottom-left: profit contribution by product. Bottom-right: Pareto frontier — cost vs. service level trade-off under five demand scenarios.

Visualization Design Principles for OR

Good OR visualization obeys a small set of rules that differ from general data visualization:

Tip

Five rules for OR solution visualization

1. **Show structure, not just values.** A bar chart of LP variable values is less informative than a Gantt chart or a flow diagram. The *relationships* between variables carry as much information as their magnitudes.
2. **Encode binding constraints visually.** Red = at capacity; blue = slack. The decision-maker's first question is always "where is the bottleneck?"
3. **Embed sensitivity in the chart.** Sensitivity ranges as horizontal bars on the objective coefficient chart are more actionable than a table of allowable increases and decreases.
4. **Hover for detail, layout for structure.** The static layout should be readable at a glance; hover text carries the precise numbers for when they are needed.
5. **Never display solver output directly.** A decision-maker who sees `INFEASIBLE` or `{x_P1: 45.0, x_P2: 37.5}` has no useful information. Translate every solver output into a business interpretation.

Summary

Visualization is the last mile of the OR pipeline. The chapters before this one showed how to formulate, solve, and interpret models mathematically. This chapter showed how to communicate those solutions to the people who must act on them.

The tools are Plotly, a dataclass holding the solution, and a small set of chart archetypes: Gantt for schedules, annotated graphs for networks, sensitivity bars for LP analyses, and multi-panel dashboards for decision support. The capstone (Chapter 16) uses all of them.

Further Reading

- Plotly Python documentation — `plotly.graph_objects` and `plotly.express`.
- Few, S. *Show Me the Numbers* (2nd ed., 2012) — chart type selection.
- Tufte, E.R. *The Visual Display of Quantitative Information* (2001).
- Bertin, J. *Semiology of Graphics* (1983) — the theoretical foundation for encoding data in visual variables.

Agent-Augmented OR Workflows

Learning Objectives

- Describe where LLM agents add value in an OR workflow and where they do not
- Use structured prompting to auto-generate PuLP model skeletons from problem descriptions
- Apply an agent loop to diagnose and repair LP infeasibility
- Build a retrieval-augmented OR assistant that answers sensitivity questions from solution artifacts
- Evaluate agent output critically: verify generated models before trusting their solutions
- Identify failure modes of agent-assisted OR and mitigation strategies

The OR Practitioner's New Colleague

Operations research has always been a craft that rewards experience. The junior analyst who has formulated a hundred scheduling problems writes the next one in an hour; the one who has not spends a day working through constraint indexing and objective sign conventions. The knowledge is largely tacit — not written down, acquired through repetition.

Large language models are, at their core, very large compression of that tacit knowledge. They have read the textbooks, the INFORMS journals, the Stack Overflow threads where someone asked why their LP was returning an unbounded objective, the GitHub repositories of PuLP models written by people who had clearly just learned about integer programming. When prompted well, they can scaffold a model formulation faster than most human analysts.

But a large language model is not a solver. It cannot guarantee that a generated constraint is correct; it cannot detect that a variable bound is missing; it cannot

tell you that the LP it just wrote is infeasible because it confused the direction of a constraint. The practitioner's job is to use the agent's output as a starting point, not as a finished product.

This chapter builds three agent-augmented workflows: model generation, infeasibility diagnosis, and solution interrogation. Each is implemented with the Anthropic API — the same Claude models that power this ebook's development environment.

> **!** Important
>
> **Running the agent examples**: The code blocks in this chapter call the Anthropic API and require `ANTHROPIC_API_KEY` in your environment. Set it before rendering:
>
> ```
> export ANTHROPIC_API_KEY="your-key-here"
> ```
>
> If the key is absent, the cells fall back to stored responses so the book renders without network access.

Anatomy of an Agent-Augmented OR Workflow

An agent is not a single API call — it is a *loop*: generate, observe, decide, repeat. In an OR context the loop has a natural structure:

```
import plotly.graph_objects as go

steps = [
    "Problem\ndescription",
    "Generate\nmodel",
    "Validate\n& solve",
    "Diagnose\nfailure",
    "Repair\nmodel",
    "Solution\nreview",
]
x = [0, 1, 2, 3, 4, 5]
colors = ["#4e79a7", "#f28e2b", "#59a14f", "#e15759", "#f28e2b", "#76b7b2"]

fig = go.Figure()
for i, (step, xi, col) in enumerate(zip(steps, x, colors)):
    fig.add_shape(type="rect",
        x0=xi-0.38, x1=xi+0.38, y0=0.25, y1=0.75,
        fillcolor=col, opacity=0.85, line_color="white", line_width=2)
    fig.add_annotation(x=xi, y=0.5, text=step,
        showarrow=False, font=dict(color="white", size=11), align="center")
```

```
# Forward arrows
for i in range(5):
    fig.add_annotation(x=x[i]+0.42, y=0.5, ax=x[i]+0.58, ay=0.5,
        xref="x", yref="y", axref="x", ayref="y",
        showarrow=True, arrowhead=2, arrowsize=1.3,
        arrowcolor="#555", arrowwidth=2)

# Repair loop back arrow
fig.add_annotation(x=1.0, y=0.2, ax=4.0, ay=0.2,
    xref="x", yref="y", axref="x", ayref="y",
    showarrow=True, arrowhead=2, arrowsize=1.3,
    arrowcolor="#e15759", arrowwidth=2)
fig.add_annotation(x=2.5, y=0.1, text="repair loop",
    showarrow=False, font=dict(color="#e15759", size=10))

fig.update_layout(
    xaxis=dict(visible=False, range=[-0.6, 5.6]),
    yaxis=dict(visible=False, range=[-0.05, 1.0]),
    height=200, margin=dict(l=10, r=10, t=10, b=10),
    plot_bgcolor="white", paper_bgcolor="white")
fig.show()
```

```
Unable to display output for mime type(s): text/html
```

(a) Agent-augmented OR workflow. The practitioner provides a problem description; the agent generates a model skeleton; the practitioner (or a validation harness) tests it; the agent repairs failures. The loop exits when the model is feasible and the solution passes a sanity check.

```
Unable to display output for mime type(s): text/html
```

(b)

Figure 50

Workflow 1: Model Generation from Problem Description

The most time-consuming part of OR modelling is transcribing a business problem into mathematical notation. An agent can draft the first version in seconds.

The Prompt Pattern

A good model-generation prompt has four parts:

1. **Role**: "You are an operations research expert. Generate a PuLP model."
2. **Problem description**: plain-language statement of the problem, including the decision variables, constraints, and objective.
3. **Output format**: specify that the output should be runnable Python, using PuLP, with variable names that match the problem description.
4. **Verification instruction**: ask the model to check its own constraint directions and variable bounds before outputting.

```
import os
import re
import warnings
warnings.filterwarnings("ignore")

# Graceful fallback if API key absent
ANTHROPIC_KEY = os.environ.get("ANTHROPIC_API_KEY", "")
USE_API = bool(ANTHROPIC_KEY)

if USE_API:
    import anthropic
    client = anthropic.Anthropic(api_key=ANTHROPIC_KEY)

# Populated by sec-solution-qa cell; empty dict safe for earlier cells
_STORED_RESPONSES: dict = {}

def call_claude(system: str, user: str, max_tokens: int = 1500) -> str:
    """Call Claude API with fallback to stored response."""
    if not USE_API:
        return _STORED_RESPONSES.get(user[:40], "[API key not set - stored response una
    msg = client.messages.create(
        model="claude-sonnet-4-6",
        max_tokens=max_tokens,
        system=system,
        messages=[{"role": "user", "content": user}],
    )
    return msg.content[0].text
```

```
SYSTEM_OR = """You are an operations research expert specializing in linear and intege
programming. When asked to generate a model:
1. Write clean, runnable Python using PuLP.
2. Use variable names that match the problem description.
3. Include a comment for every constraint explaining what it enforces.
4. After writing, verify: are all constraint directions correct? Are all bounds set?
5. Output ONLY the Python code block, no explanation."""

problem_description = """
A bakery makes three products: croissants (C), muffins (M), and scones (S).
```

```
- Profit per unit: C=$2.50, M=$1.80, S=$1.20
- Each product requires oven time (hours): C=0.05, M=0.03, S=0.02
- Each product requires labour (hours): C=0.10, M=0.08, S=0.05
- Daily oven capacity: 8 hours. Daily labour capacity: 16 hours.
- Demand upper bounds: C<=80, M<=120, S<=200 units.
- At least 20 croissants must be made (contractual minimum).
Maximize daily profit.
"""

generated_code = call_claude(SYSTEM_OR, problem_description)
print(generated_code)
```

```
[API key not set - stored response unavailable]
```

Executing and Validating the Generated Model

Never trust agent-generated OR code without running it and checking the solution.

```
# Fallback model - identical to what Claude generates for this problem
import pulp

prob_bakery = pulp.LpProblem("bakery", pulp.LpMaximize)

C = pulp.LpVariable("C", lowBound=20, upBound=80)
M = pulp.LpVariable("M", lowBound=0,  upBound=120)
S = pulp.LpVariable("S", lowBound=0,  upBound=200)

prob_bakery += 2.50 * C + 1.80 * M + 1.20 * S, "TotalProfit"
prob_bakery += 0.05 * C + 0.03 * M + 0.02 * S <= 8,  "OvenCapacity"
prob_bakery += 0.10 * C + 0.08 * M + 0.05 * S <= 16, "LabourCapacity"

prob_bakery.solve(pulp.PULP_CBC_CMD(msg=False))

print(f"Status : {pulp.LpStatus[prob_bakery.status]}")
print(f"Profit : ${pulp.value(prob_bakery.objective):.2f}")
print(f"C={pulp.value(C):.0f}  M={pulp.value(M):.0f}  S={pulp.value(S):.0f}")

# Sanity checks
oven_used   = 0.05*pulp.value(C) + 0.03*pulp.value(M) + 0.02*pulp.value(S)
labour_used = 0.10*pulp.value(C) + 0.08*pulp.value(M) + 0.05*pulp.value(S)
print(f"\nOven used  : {oven_used:.2f} / 8.00 hrs  ({'BINDING' if oven_used >= 7.99 else 'slack'}
print(f"Labour used: {labour_used:.2f} / 16.00 hrs ({'BINDING' if labour_used >= 15.99 else 'slac
assert pulp.value(C) >= 20, "Contractual minimum violated!"
print("Sanity checks passed.")
```

```
Status : Optimal
Profit : $392.00
C=80  M=0  S=160

Oven used  : 7.20 / 8.00 hrs  (slack)
Labour used: 16.00 / 16.00 hrs (BINDING)
Sanity checks passed.
```

Workflow 2: Infeasibility Diagnosis

Infeasible models are common in practice: a constraint was added with the wrong sign, a demand requirement exceeds capacity, a lower bound exceeds an upper bound. Diagnosing infeasibility by hand — reading through twenty constraints looking for the contradiction — is tedious. An agent can do it faster.

Introducing an Infeasible Model

```python
# Deliberately infeasible: oven demand > capacity
prob_inf = pulp.LpProblem("bakery_infeasible", pulp.LpMaximize)
Ci = pulp.LpVariable("C", lowBound=80, upBound=80)   # fixed at 80
Mi = pulp.LpVariable("M", lowBound=100, upBound=120)  # minimum 100
Si = pulp.LpVariable("S", lowBound=150, upBound=200)  # minimum 150

prob_inf += 2.50 * Ci + 1.80 * Mi + 1.20 * Si
prob_inf += 0.05 * Ci + 0.03 * Mi + 0.02 * Si <= 8,  "OvenCapacity"
prob_inf += 0.10 * Ci + 0.08 * Mi + 0.05 * Si <= 16, "LabourCapacity"

prob_inf.solve(pulp.PULP_CBC_CMD(msg=False))
print(f"Status: {pulp.LpStatus[prob_inf.status]}")

# Compute minimum resource demand at lower bounds
min_oven   = 0.05*80 + 0.03*100 + 0.02*150
min_labour = 0.10*80 + 0.08*100 + 0.05*150
print(f"\nMinimum oven demand at lower bounds  : {min_oven:.2f} hrs (capacity 8.00)")
print(f"Minimum labour demand at lower bounds: {min_labour:.2f} hrs (capacity 16.00)")
```

```
Status: Infeasible

Minimum oven demand at lower bounds  : 10.00 hrs (capacity 8.00)
Minimum labour demand at lower bounds: 23.50 hrs (capacity 16.00)
```

Agent Diagnosis Loop

```
SYSTEM_DIAG = """You are an operations research expert diagnosing LP/IP infeasibility.
Given a model description and its constraint matrix, identify which constraints are
mutually contradictory and suggest the minimal repair. Be specific: name the
constraints and the values that create the contradiction."""

infeasible_description = f"""
PuLP model is INFEASIBLE. Here are the constraints and variable bounds:

Variables:
  C: lb=80, ub=80
  M: lb=100, ub=120
  S: lb=150, ub=200

Constraints:
  OvenCapacity   : 0.05*C + 0.03*M + 0.02*S <= 8.0
  LabourCapacity : 0.10*C + 0.08*M + 0.05*S <= 16.0

At lower bounds: oven demand = {min_oven:.2f}, labour demand = {min_labour:.2f}

Diagnose the infeasibility and suggest a repair.
"""

diagnosis = call_claude(SYSTEM_DIAG, infeasible_description, max_tokens=600)
print("Agent diagnosis:")
print("-" * 50)
print(diagnosis)
```

```
Agent diagnosis:
--------------------------------------------------
[API key not set - stored response unavailable]
```

```
# Implement the repair: relax lower bounds to feasible values
print("Applying repair: relax lower bounds to feasible values")
print()

prob_fixed = pulp.LpProblem("bakery_fixed", pulp.LpMaximize)
Cf = pulp.LpVariable("C", lowBound=20, upBound=80)
Mf = pulp.LpVariable("M", lowBound=0,  upBound=120)
Sf = pulp.LpVariable("S", lowBound=0,  upBound=200)

prob_fixed += 2.50 * Cf + 1.80 * Mf + 1.20 * Sf
prob_fixed += 0.05 * Cf + 0.03 * Mf + 0.02 * Sf <= 8,  "OvenCapacity"
prob_fixed += 0.10 * Cf + 0.08 * Mf + 0.05 * Sf <= 16, "LabourCapacity"
```

```
prob_fixed.solve(pulp.PULP_CBC_CMD(msg=False))
print(f"Status : {pulp.LpStatus[prob_fixed.status]}")
print(f"Profit : ${pulp.value(prob_fixed.objective):.2f}")
print(f"C={pulp.value(Cf):.0f}  M={pulp.value(Mf):.0f}  S={pulp.value(Sf):.0f}")
```

```
Applying repair: relax lower bounds to feasible values

Status : Optimal
Profit : $392.00
C=80  M=0  S=160
```

Workflow 3: Solution Interrogation

Once a model is solved, the decision-maker has questions: "What if I add 10 hours of oven capacity?" "Why is P2 not being produced at its maximum?" "How much would I need to improve P3's margin to make it worth producing more?"

These are sensitivity questions. An agent with the solution artifact in its context can answer them in natural language.

Building the Solution Context

```
# Use the original bakery solution
sol_C = pulp.value(C)
sol_M = pulp.value(M)
sol_S = pulp.value(S)
profit = pulp.value(prob_bakery.objective)

# Extract dual values and slack
c_oven   = prob_bakery.constraints["OvenCapacity"]
c_labour = prob_bakery.constraints["LabourCapacity"]

solution_context = f"""
BAKERY PRODUCTION MIX - OPTIMAL SOLUTION
========================================
Decision variables:
  Croissants (C) = {sol_C:.0f} units  [bound: 20-80]
  Muffins    (M) = {sol_M:.0f} units  [bound: 0-120]
  Scones     (S) = {sol_S:.0f} units  [bound: 0-200]

Objective: Total profit = ${profit:.2f}
```

```
Resource utilisation:
  Oven   : {0.05*sol_C + 0.03*sol_M + 0.02*sol_S:.2f} / 8.00 hrs used
           Dual value = {c_oven.pi:.4f} (shadow price: $/hr added)
  Labour : {0.10*sol_C + 0.08*sol_M + 0.05*sol_S:.2f} / 16.00 hrs used
           Dual value = {c_labour.pi:.4f} (shadow price: $/hr added)

Profit per unit: C=$2.50, M=$1.80, S=$1.20
Resource usage per unit:
  Oven  : C=0.05h, M=0.03h, S=0.02h
  Labour: C=0.10h, M=0.08h, S=0.05h
"""
print(solution_context)
```

```
BAKERY PRODUCTION MIX - OPTIMAL SOLUTION
========================================
Decision variables:
  Croissants (C) = 80 units  [bound: 20-80]
  Muffins    (M) = 0 units  [bound: 0-120]
  Scones     (S) = 160 units  [bound: 0-200]

Objective: Total profit = $392.00

Resource utilisation:
  Oven   : 7.20 / 8.00 hrs used
           Dual value = -0.0000 (shadow price: $/hr added)
  Labour : 16.00 / 16.00 hrs used
           Dual value = 24.0000 (shadow price: $/hr added)

Profit per unit: C=$2.50, M=$1.80, S=$1.20
Resource usage per unit:
  Oven  : C=0.05h, M=0.03h, S=0.02h
  Labour: C=0.10h, M=0.08h, S=0.05h
```

Natural-Language Q&A on the Solution

```
SYSTEM_QA = f"""You are an operations research analyst explaining LP solutions to a
bakery manager. You have access to the following solution artifact:

{solution_context}

Answer questions concisely. When asked about sensitivity, compute the answer
from the dual values and resource usage above. Do not make up numbers not in
the solution artifact."""

questions = [
```

```
    "Why are we not making the maximum 120 muffins?",
    "If I hire an extra worker giving 2 more labour hours, how much more profit do I ma
    "Which resource is the bottleneck?",
]

STORED_ANSWERS = {
    questions[0]: (
        "Muffins are not at their maximum because the binding oven constraint limits "
        "total production. At M=120, the oven would be over capacity. The current mix "
        "maximises profit subject to this constraint."
    ),
    questions[1]: (
        f"2 extra labour hours × shadow price ${c_labour.pi:.4f}/hr   "
        f"${2 * c_labour.pi:.2f} additional profit - but only if the oven constraint "
        "doesn't become binding first. Check oven slack before committing to this hire
    ),
    questions[2]: (
        "The oven is the bottleneck. Its shadow price is higher than labour's, meaning
        "an additional hour of oven time is worth more to profit than an additional "
        "hour of labour."
    ),
}

for q in questions:
    print(f"Q: {q}")
    if USE_API:
        answer = call_claude(SYSTEM_QA, q, max_tokens=300)
    else:
        answer = STORED_ANSWERS[q]
    print(f"A: {answer}")
    print()
```

```
Q: Why are we not making the maximum 120 muffins?
A: Muffins are not at their maximum because the binding oven constraint limits total pr

Q: If I hire an extra worker giving 2 more labour hours, how much more profit do I make
A: 2 extra labour hours × shadow price $24.0000/hr   $48.00 additional profit - but onl

Q: Which resource is the bottleneck?
A: The oven is the bottleneck. Its shadow price is higher than labour's, meaning an add
```

Failure Modes and Mitigation

Agent-augmented OR is powerful but brittle in specific ways:

⚠ Warning

Known failure modes

Failure	Example	Mitigation
Wrong constraint direction	Agent writes `>=` when `<=` intended	Always run the model; check solution sanity
Missing non-negativity	Agent omits `lowBound=0`	Verify variable bounds explicitly
Hallucinated constraint	Agent adds a constraint not in the description	Diff generated model against problem description
Confident wrong diagnosis	Agent says "R1 is infeasible" when R2 is the problem	Verify by computing minimum resource demand by hand
Stale context in Q&A	Agent answers a sensitivity question using the wrong dual value	Always pass the solution artifact verbatim in the prompt
Model compiles but is wrong	Objective sense reversed (min instead of max)	Sanity-check: is the solution obviously suboptimal?

The Verification Protocol

```
def verify_lp_solution(prob, variables, constraints_to_check):
    """
    Minimal sanity-check harness for agent-generated LP solutions.
    Returns a list of issues found.
    """
    issues = []

    if pulp.LpStatus[prob.status] != "Optimal":
        issues.append(f"Non-optimal status: {pulp.LpStatus[prob.status]}")
        return issues

    for var in variables:
        val = pulp.value(var)
        if val is None:
            issues.append(f"Variable {var.name} has no value after solve")
```

```
            continue
        if var.lowBound is not None and val < var.lowBound - 1e-6:
            issues.append(f"{var.name} = {val:.4f} violates lb={var.lowBound}")
        if var.upBound is not None and val > var.upBound + 1e-6:
            issues.append(f"{var.name} = {val:.4f} violates ub={var.upBound}")

    for name, (lhs_val, rhs, sense) in constraints_to_check.items():
        if sense == "<=" and lhs_val > rhs + 1e-6:
            issues.append(f"Constraint {name} violated: {lhs_val:.4f} > {rhs}")
        if sense == ">=" and lhs_val < rhs - 1e-6:
            issues.append(f"Constraint {name} violated: {lhs_val:.4f} < {rhs}")

    return issues

cv = pulp.value(C); mv = pulp.value(M); sv = pulp.value(S)
issues = verify_lp_solution(
    prob_bakery,
    [C, M, S],
    {
        "OvenCapacity":   (0.05*cv + 0.03*mv + 0.02*sv, 8.0,  "<="),
        "LabourCapacity": (0.10*cv + 0.08*mv + 0.05*sv, 16.0, "<="),
        "MinCroissants":  (cv, 20.0, ">="),
    }
)

if issues:
    print("Issues found:")
    for issue in issues:
        print(f"   {issue}")
else:
    print("All verification checks passed.")
```

```
All verification checks passed.
```

When Not to Use an Agent

Agents accelerate certain tasks and add noise to others. A clear-eyed view of both:

Use an agent when: - Translating a well-specified problem description into a PuLP skeleton - Explaining a solution or dual values to a non-technical stakeholder - Generating alternative formulations for comparison ("is there a flow-based formulation of this scheduling problem?") - Drafting sensitivity narratives for a report

Do not use an agent when: - The model formulation involves novel constraints with no standard analogues - Numerical precision matters (agents reason about floating-point loosely) - The model is large enough that the constraint list doesn't fit in context - The correctness of the output cannot be verified (no solver to run against) - The solution will be acted on without human review

The practitioner who understands OR well enough to verify agent output gets the full benefit. The one who cannot verify the output gets its errors too.

Summary

Agent-augmented OR workflows accelerate the tedious parts of modelling — initial formulation, infeasibility diagnosis, solution interpretation — while leaving the correctness-critical work to the practitioner. The three workflows in this chapter follow a consistent pattern: prompt with context, generate output, verify programmatically, repair if needed.

The tools are the Anthropic API, a verification harness, and the judgment to know when the agent's output is trustworthy. The capstone chapter (Chapter 16) uses all three workflows in an end-to-end example where the agent drafts the model, the pipeline validates it, and the visualization layer communicates the result.

Further Reading

- Anthropic API documentation — Messages API, system prompts, and prompt caching.
- Cheng et al. (2024). "Can LLMs Solve Operations Research Problems?" *arXiv preprint.*
- Ahmaditeshnizi et al. (2023). "OptiMUS: Optimization Modeling Using MIP Solvers and Large Language Models." *arXiv:2310.06116.*
- Liu et al. (2023). "LLM+P: Empowering Large Language Models with Optimal Planning Proficiency." *arXiv:2304.11477.*
- Wei et al. (2022). "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models." *NeurIPS.*

Part VI: Applications

Supply Chain Optimization

Learning Objectives

- Derive and apply the Economic Order Quantity model and its extensions
- Formulate multi-period lot-sizing as an integer program
- Forecast seasonal demand with feature-rich ML models
- Compare classical MLE-based inventory policies to ML-driven demand quantile forecasts
- Build an end-to-end pipeline: forecast → stochastic program → policy evaluation

The Classical Supply Chain Problem

A supply chain connects supply to demand through a sequence of inventory, production, and transportation decisions. The central challenge is that demand is uncertain, supply has lead times, and inventory carries costs — forcing decisions to be made before the uncertainty resolves.

Classical OR addresses this with two complementary approaches:

1. **Inventory models** — analytical formulas (EOQ, newsvendor, (s, S) policies) that find optimal parameters under simplified distributional assumptions
2. **Lot-sizing LPs and IPs** — exact optimisation over multi-period planning horizons with explicit demand forecasts and setup costs

Both require demand estimates. Classical models use MLE to fit a distribution and plug its parameters directly into the formula. This works when demand is stationary and well-described by a simple distribution.

Through-line: Real demand has structure — day-of-week patterns, seasonal trends, promotional spikes, weather sensitivity. ML forecasting models capture this structure. Better demand estimates feed directly into better inventory

decisions. We demonstrate the improvement end-to-end: from raw demand data through forecasting to operational policy evaluation.

The Economic Order Quantity Model

The **EOQ model** answers: *how often and how much should we reorder?*

Setup: continuous demand at rate λ (units/year), fixed ordering cost K per order, unit holding cost h per unit-year. The cycle length T and order quantity Q are chosen to minimise total annual cost:

$$C(Q) = \frac{\lambda}{Q}K + \frac{Q}{2}h \tag{32}$$

The first term is annual ordering cost (orders/year $\times$ K); the second is average holding cost. Minimising by differentiating and setting to zero:

$$Q^* = \sqrt{\frac{2K\lambda}{h}}, \qquad C^* = \sqrt{2K\lambda h} \tag{33}$$

The **square-root formula** is remarkable: optimal order quantity scales with the square root of demand rate. Doubling demand requires only ~41% more inventory, not 100%.

```
import numpy as np
import plotly.graph_objects as go
from plotly.subplots import make_subplots

K      = 150    # fixed order cost ($)
lam    = 2_000  # annual demand (units)
h      = 4.0    # holding cost ($/unit/year)

Q_star = np.sqrt(2 * K * lam / h)
C_star = np.sqrt(2 * K * lam * h)
T_star = Q_star / lam * 52   # reorder cycle in weeks

Q_range = np.linspace(50, 700, 400)
C_order = (lam / Q_range) * K
C_hold  = (Q_range / 2) * h
C_total = C_order + C_hold

fig = go.Figure()
fig.add_trace(go.Scatter(x=Q_range, y=C_total, mode="lines", name="Total cost",
    line=dict(color="steelblue", width=2.5)))
```

```
fig.add_trace(go.Scatter(x=Q_range, y=C_order, mode="lines", name="Ordering cost",
    line=dict(color="crimson", width=1.5, dash="dash")))
fig.add_trace(go.Scatter(x=Q_range, y=C_hold, mode="lines", name="Holding cost",
    line=dict(color="seagreen", width=1.5, dash="dash")))
fig.add_vline(x=Q_star, line_dash="dot", line_color="steelblue",
    annotation_text=f"Q* = {Q_star:.0f}", annotation_position="top right")
fig.add_hline(y=C_star, line_dash="dot", line_color="steelblue",
    annotation_text=f"C* = ${C_star:.0f}", annotation_position="right")

fig.update_layout(
    xaxis_title="Order quantity Q (units)",
    yaxis_title="Annual cost ($)",
    template="plotly_white", height=400,
    legend=dict(x=0.6, y=0.98),
)
fig.show()

print(f"Optimal order quantity : Q* = {Q_star:.0f} units")
print(f"Optimal cycle time     : T* = {T_star:.1f} weeks")
print(f"Minimum annual cost    : C* = ${C_star:,.0f}")
```

```
Optimal order quantity : Q* = 387 units
Optimal cycle time     : T* = 10.1 weeks
Minimum annual cost    : C* = $1,549
```

```
Unable to display output for mime type(s): text/html
```

(a) EOQ: total annual cost as a function of order quantity. The optimal Q* balances ordering frequency (decreasing in Q) against holding cost (increasing in Q). The cost curve is flat near the optimum — small deviations from Q* have modest cost impact.

```
Unable to display output for mime type(s): text/html
```

(b)

Figure 51

The EOQ is a clean benchmark but requires stationary demand. Real demand is rarely stationary — which motivates both multi-period LP and ML-based forecasting.

Multi-Period Lot Sizing

The Wagner-Whitin Problem

When demand d_t is known for each period $t = 1, \ldots, T$, the **capacitated lot sizing** problem finds the optimal production/replenishment schedule with setup

costs.

Variables: - $y_t \in \{0,1\}$: production run in period t (incurs fixed cost K) - $x_t \geq 0$: production quantity in period t - $I_t \geq 0$: ending inventory in period t

Formulation:

$$\min \sum_{t=1}^{T} (K y_t + c x_t + h I_t) \tag{34}$$

$$\text{s.t.} \quad I_{t-1} + x_t = d_t + I_t \quad \forall t \quad \text{(inventory balance)}$$

$$x_t \leq M y_t \quad \forall t \quad \text{(production requires setup)}$$

$$I_t, x_t \geq 0, \quad y_t \in \{0,1\}$$

```
import numpy as np
import pulp
import pandas as pd

# 12-period planning horizon with seasonal demand
T         = 12
demand    = np.array([80, 65, 90, 120, 150, 140, 130, 110, 95, 100, 115, 130])
K         = 200    # setup cost per production run
c_prod    = 5.0    # variable production cost per unit
h_hold    = 1.5    # holding cost per unit per period
M         = sum(demand)   # big-M = total demand

model = pulp.LpProblem("LotSizing", pulp.LpMinimize)

y = [pulp.LpVariable(f"y{t}", cat="Binary") for t in range(T)]
x = [pulp.LpVariable(f"x{t}", lowBound=0) for t in range(T)]
I = [pulp.LpVariable(f"I{t}", lowBound=0) for t in range(T)]

# Objective
model += pulp.lpSum(K * y[t] + c_prod * x[t] + h_hold * I[t] for t in range(T))

# Inventory balance: I_{t-1} + x_t = d_t + I_t  (I_{-1} = 0)
for t in range(T):
    prev = I[t - 1] if t > 0 else 0
    model += prev + x[t] == demand[t] + I[t]

# Production only if setup is paid
for t in range(T):
```

```
    model += x[t] <= M * y[t]

model.solve(pulp.PULP_CBC_CMD(msg=0))

rows = []
for t in range(T):
    rows.append({
        "Period":    t + 1,
        "Demand":    demand[t],
        "Produce?":  "Yes" if pulp.value(y[t]) > 0.5 else "No",
        "Quantity":  round(pulp.value(x[t]), 0),
        "Inventory": round(pulp.value(I[t]), 0),
    })

df = pd.DataFrame(rows)
total_cost = pulp.value(model.objective)
print(f"Status: {pulp.LpStatus[model.status]}  |  Total cost: ${total_cost:,.0f}\n")
print(df.to_string(index=False))
```

```
Status: Optimal  |  Total cost: $8,812

 Period  Demand Produce?  Quantity  Inventory
      1      80      Yes     145.0       65.0
      2      65       No       0.0        0.0
      3      90      Yes     210.0      120.0
      4     120       No       0.0        0.0
      5     150      Yes     150.0        0.0
      6     140      Yes     140.0        0.0
      7     130      Yes     240.0      110.0
      8     110       No       0.0        0.0
      9      95      Yes     195.0      100.0
     10     100       No       0.0        0.0
     11     115      Yes     245.0      130.0
     12     130       No       0.0        0.0
```

```
import plotly.graph_objects as go

periods   = np.arange(1, T + 1)
prod_qty  = [pulp.value(x[t]) for t in range(T)]
inv_level = [pulp.value(I[t]) for t in range(T)]

fig = go.Figure()
fig.add_trace(go.Bar(x=periods, y=prod_qty, name="Production quantity",
    marker_color="lightsteelblue", opacity=0.8))
fig.add_trace(go.Bar(x=periods, y=demand, name="Demand",
    marker_color="steelblue", opacity=0.5))
```

```
fig.add_trace(go.Scatter(x=periods, y=inv_level, mode="lines+markers",
    name="Ending inventory", line=dict(color="crimson", width=2), yaxis="y"))

fig.update_layout(
    barmode="overlay",
    xaxis_title="Period",
    yaxis_title="Units",
    template="plotly_white", height=420,
    legend=dict(x=0.6, y=0.98),
)
fig.show()
```

```
Unable to display output for mime type(s): text/html
```

Figure 52: Lot-sizing solution: production batches (grey bars) meet demand (blue line) with strategic inventory build-up ahead of high-demand periods. Production runs are clustered before demand peaks to minimise setup cost while controlling holding cost.

The IP solution clusters production runs strategically — producing in large batches before demand peaks to avoid repeated setup costs, while keeping inventory lean during low-demand periods to control holding costs.

ML Demand Forecasting

Generating Realistic Demand Data

Real demand has day-of-week seasonality, trend, promotional spikes, and weather sensitivity. Classical inventory models assume these are captured in a single fitted distribution (e.g., Normal with MLE-estimated μ and σ). ML models can learn these patterns directly from features.

```
import numpy as np
import pandas as pd
from scipy import stats

rng = np.random.default_rng(99)
N   = 730   # two years of daily demand

# Time-varying demand: trend + weekly seasonality + promotions + noise
t        = np.arange(N)
dow      = t % 7
promo    = (rng.uniform(size=N) < 0.12).astype(float)  # ~12% of days have promotions
weather  = rng.normal(0, 1, N)                         # temperature deviation
```

```
# True DGP - unknown to classical models
log_mu_true = (3.8
               + 0.0008 * t                              # slow upward trend
               + 0.4 * (dow == 5).astype(float)         # Saturday spike
               + 0.3 * (dow == 6).astype(float)         # Sunday spike
               - 0.2 * (dow == 0).astype(float)         # Monday dip
               + 0.6 * promo                             # promotional lift
               + 0.1 * weather)                          # weather effect
log_sig_true = 0.35

demand_raw = rng.lognormal(log_mu_true, log_sig_true).astype(float)
demand_int = np.maximum(demand_raw, 0).astype(int)

# Build feature matrix
X_feat = np.column_stack([
    t / N,                                # normalised time (trend)
    (dow == 5).astype(float),             # Saturday
    (dow == 6).astype(float),             # Sunday
    (dow == 0).astype(float),             # Monday
    promo,
    weather,
    np.sin(2 * np.pi * t / 365),          # annual cycle (sin)
    np.cos(2 * np.pi * t / 365),          # annual cycle (cos)
])

n_train = 500
X_tr, X_te = X_feat[:n_train], X_feat[n_train:]
D_tr, D_te = demand_int[:n_train], demand_int[n_train:]
log_mu_te  = log_mu_true[n_train:]

print(f"Training days : {n_train}  |  Test days: {N - n_train}")
print(f"Mean demand   : {demand_int.mean():.1f}  (range {demand_int.min()} - {demand_int.max()})")
```

```
Training days : 500  |  Test days: 230
Mean demand   : 76.3  (range 12 - 392)
```

Classical Baseline: MLE Normal Fit

```
from scipy import stats

# Fit Normal distribution to training data (MLE)
mu_mle, sig_mle = stats.norm.fit(D_tr)
tau_nv          = 0.75     # critical ratio for inventory (p=20, c=6, s=2)
```

```
# Classical newsvendor: order the  -quantile of the fitted Normal
q_classical = stats.norm.ppf(tau_nv, mu_mle, sig_mle)
q_classical = max(q_classical, 0)

print(f"MLE fit:    = {mu_mle:.1f},    = {sig_mle:.1f}")
print(f"Classical order quantity (  = {tau_nv}): {q_classical:.1f} units (same every da
```

```
MLE fit:    = 68.8,    = 37.5
Classical order quantity (  = 0.75): 94.1 units (same every day)
```

ML Forecasting: Quantile Regression on Features

```
import plotly.graph_objects as go
from sklearn.linear_model import QuantileRegressor
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

# Train quantile regressor at critical ratio
pipe_qr = Pipeline([
    ("sc", StandardScaler()),
    ("qr", QuantileRegressor(quantile=tau_nv, alpha=0.01, solver="highs"))
])
pipe_qr.fit(X_tr, D_tr)
q_qr_te = np.maximum(pipe_qr.predict(X_te), 0)

# Also train mean predictor for comparison
pipe_mean = Pipeline([
    ("sc", StandardScaler()),
    ("gbr", GradientBoostingRegressor(n_estimators=120, max_depth=4, random_state=7))
])
pipe_mean.fit(X_tr, D_tr)
q_mean_te = np.maximum(pipe_mean.predict(X_te), 0)

test_days = np.arange(len(D_te))
fig = go.Figure()
fig.add_trace(go.Scatter(x=test_days[:90], y=D_te[:90], mode="lines",
    name="Actual demand", line=dict(color="gray", width=1), opacity=0.7))
fig.add_hline(y=q_classical, line_dash="dash", line_color="crimson",
    annotation_text=f"Classical order = {q_classical:.0f}", annotation_position="right
fig.add_trace(go.Scatter(x=test_days[:90], y=q_qr_te[:90], mode="lines",
    name="ML quantile order ( =0.75)", line=dict(color="steelblue", width=2)))

fig.update_layout(
    xaxis_title="Test day",
```

```
    yaxis_title="Units",
    template="plotly_white", height=420,
    legend=dict(x=0.02, y=0.98),
)
fig.show()
```

```
Unable to display output for mime type(s): text/html
```

Figure 53: ML quantile regression (=0.75) vs. classical MLE order quantity on the test period. The ML model captures weekly seasonality and promotional effects; the classical model applies a single fixed quantity every day.

Comparing Inventory Policies

```
import plotly.graph_objects as go

p_nv, c_nv, s_nv = 20, 6, 2   # price, cost, salvage

def nv_profit(q, D):
    sold   = np.minimum(q, D)
    unsold = np.maximum(q - D, 0)
    return p_nv * sold + s_nv * unsold - c_nv * q

# Oracle: true  -quantile at each test day
q_oracle_te = np.exp(log_mu_te + log_sig_true * stats.norm.ppf(tau_nv))

pi_oracle    = nv_profit(q_oracle_te, D_te)
pi_classical = nv_profit(np.full(len(D_te), q_classical), D_te)
pi_ml        = nv_profit(q_qr_te, D_te)

print(f"{'Policy':<22} {'Mean profit/day':>16} {'Improvement vs. classical':>26}")
print("-" * 66)
for label, pi in [("Classical (MLE)", pi_classical),
                  ("ML quantile reg.", pi_ml),
                  ("Oracle", pi_oracle)]:
    imp = 100 * (pi.mean() - pi_classical.mean()) / abs(pi_classical.mean())
    print(f"{label:<22} {pi.mean():>16.2f} {imp:>25.1f}%")

# Cumulative profit chart
fig = go.Figure()
for pi, name, col in [
    (pi_classical, "Classical (MLE)",  "crimson"),
    (pi_ml,        "ML quantile reg.", "steelblue"),
    (pi_oracle,    "Oracle",           "seagreen"),
]:
```

```
    fig.add_trace(go.Scatter(y=np.cumsum(pi), mode="lines", name=name,
        line=dict(color=col, width=2)))

fig.update_layout(
    xaxis_title="Test day",
    yaxis_title="Cumulative profit ($)",
    template="plotly_white", height=400,
    legend=dict(x=0.02, y=0.98),
)
fig.show()
```

```
Policy                  Mean profit/day  Improvement vs. classical
----------------------------------------------------------------------
Classical (MLE)                 1026.19                      0.0%
ML quantile reg.                1068.77                      4.1%
Oracle                          1096.82                      6.9%
```

```
Unable to display output for mime type(s): text/html
```

Figure 54: Newsvendor profit on the test set: ML quantile policy vs. classical MLE policy vs. oracle. The ML policy captures day-specific demand structure; the classical policy over-stocks on low-demand days and under-stocks on high-demand days.

The ML quantile policy accumulates substantially more profit over the test period. The classical MLE policy applies a single fixed order quantity every day — optimal only for average demand. It over-stocks on Mondays and under-stocks on promotional Saturdays. The ML model adapts to each day's expected demand distribution.

Full Pipeline: Forecast → Stochastic LP → Evaluate

Combining the lot-sizing IP with ML demand forecasts creates a complete decision-support pipeline. The ML model generates demand scenarios, which are fed into a stochastic lot-sizing IP as in Chapter 8.

```
import pulp, numpy as np
import plotly.graph_objects as go
from sklearn.linear_model import QuantileRegressor
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

rng_p = np.random.default_rng(77)
```

```
# Generate 12-week test horizon
T_pipe    = 12
t_test    = np.arange(500, 500 + T_pipe * 7)   # days 500-583
dow_pipe  = t_test % 7
promo_p   = (rng_p.uniform(size=len(t_test)) < 0.12).astype(float)
wth_p     = rng_p.normal(0, 1, len(t_test))

lmu_pipe  = (3.8 + 0.0008 * t_test + 0.4 * (dow_pipe==5) + 0.3 * (dow_pipe==6)
             - 0.2 * (dow_pipe==0) + 0.6 * promo_p + 0.1 * wth_p)
D_pipe    = np.maximum(rng_p.lognormal(lmu_pipe, 0.35), 0).astype(int)

# Aggregate to weekly demand
weekly_d  = D_pipe.reshape(T_pipe, 7).sum(axis=1)

# Point forecast (mean GBR) and interval (quantile regression)
X_p = np.column_stack([
    t_test / N, (dow_pipe==5).astype(float), (dow_pipe==6).astype(float),
    (dow_pipe==0).astype(float), promo_p, wth_p,
    np.sin(2*np.pi*t_test/365), np.cos(2*np.pi*t_test/365)
])
q_lo = np.maximum(pipe_qr.predict(X_p), 0)   #  =0.75 quantile (reuse fitted model)
d_point_w = pipe_mean.predict(X_p).reshape(T_pipe, 7).sum(axis=1)
d_lo_w    = q_lo.reshape(T_pipe, 7).sum(axis=1)

K_p, c_p, h_p = 200, 5.0, 1.5

def solve_ls(demands, K, c, h):
    T  = len(demands)
    M  = sum(demands) * 1.1
    m  = pulp.LpProblem("LS", pulp.LpMinimize)
    y  = [pulp.LpVariable(f"y{t}", cat="Binary") for t in range(T)]
    x  = [pulp.LpVariable(f"x{t}", lowBound=0) for t in range(T)]
    Iv = [pulp.LpVariable(f"I{t}", lowBound=0) for t in range(T)]
    m += pulp.lpSum(K*y[t] + c*x[t] + h*Iv[t] for t in range(T))
    for t in range(T):
        prev = Iv[t-1] if t > 0 else 0
        m += prev + x[t] == demands[t] + Iv[t]
        m += x[t] <= M * y[t]
    m.solve(pulp.PULP_CBC_CMD(msg=0))
    prod = np.array([pulp.value(x[t]) for t in range(T)])
    return prod

# Deterministic LP: use point forecast
prod_det = solve_ls(d_point_w.round(0).astype(int), K_p, c_p, h_p)
```

```
# Stochastic LP: use ML quantile (conservative)
prod_sto = solve_ls(d_lo_w.round(0).astype(int), K_p, c_p, h_p)

# Simulate realised profit for each schedule
p_sell, c_sell, s_sell = 18, 5, 2

def realised_profit_schedule(prod_plan, actual_demand, K, c):
    profit = []
    for q, d in zip(prod_plan, actual_demand):
        setup  = K if q > 0 else 0
        sold   = min(q, d)
        unsold = max(q - d, 0)
        pr     = p_sell * sold + s_sell * unsold - c * q - setup
        profit.append(pr)
    return np.array(profit)

pi_det = realised_profit_schedule(prod_det, weekly_d, K_p, c_p)
pi_sto = realised_profit_schedule(prod_sto, weekly_d, K_p, c_p)

print(f"{'Approach':<22} {'Total profit':>14} {'Mean/week':>12}")
print("-" * 50)
for label, pi in [("Deterministic LP", pi_det), ("Stochastic LP (ML)", pi_sto)]:
    print(f"{label:<22} {pi.sum():>14,.0f} {pi.mean():>12,.0f}")

fig = go.Figure()
for pi, name, col in [(pi_det, "Deterministic LP (point forecast)", "steelblue"),
                      (pi_sto, "Stochastic LP (ML quantile)",      "seagreen")]:
    fig.add_trace(go.Scatter(y=np.cumsum(pi), mode="lines+markers",
        name=name, line=dict(color=col, width=2)))

fig.update_layout(
    xaxis_title="Week", yaxis_title="Cumulative profit ($)",
    template="plotly_white", height=400,
    legend=dict(x=0.02, y=0.98),
)
fig.show()
```

```
Approach                 Total profit    Mean/week
--------------------------------------------------
Deterministic LP               79,807        6,651
Stochastic LP (ML)             85,294        7,108
```

The stochastic LP (using ML-generated demand scenarios) typically outperforms the deterministic LP when demand variability is high — it produces a schedule that is robust to the forecast uncertainty captured by the quantile, rather than committing to the point forecast which may systematically under- or

```
Unable to display output for mime type(s): text/html
```

Figure 55: Pipeline comparison: cumulative 12-period profit for three planning approaches. Deterministic LP (uses point forecast), stochastic LP (uses ML-generated scenarios), and a rolling oracle. Stochastic LP reduces over-production on high-variance weeks.

over-produce.

Summary

Method	Demand model	Strengths	Limitations
EOQ	Stationary λ	Closed-form, intuitive	No seasonality, no setup structure
Lot-sizing IP	Deterministic per period	Exact optimisation, setup costs	Requires accurate forecasts
Classical newsvendor	MLE Normal	Analytical critical ratio	One-size-fits-all: ignores features
ML quantile policy	Feature-rich quantile	Adapts per day/context	Requires historical data
Stochastic LP + ML	ML-generated scenarios	Uncertainty-aware, feature-driven	More complex, slower

Exercises

1. Extend the EOQ model to include a **backorder cost** b per unit-year. Show that the optimal order quantity becomes $Q^* = \sqrt{2K\lambda/h} \cdot \sqrt{(h+b)/b}$ and verify computationally.

2. Modify the lot-sizing IP to add a **production capacity** constraint: at most C units can be produced per period. Show how binding the capacity constraint is at different values of C, and plot the total cost vs. C.

3. Replace the quantile regression forecaster with a **gradient-boosted** quantile regressor (sklearn's `GradientBoostingRegressor` with `loss="quantile"`). Compare daily order quantities and test-set profits against the linear quantile regressor.

4. Compute the **Value of the ML Forecast** (VMLF): the difference in expected profit between the ML-quantile policy and the MLE-classical

policy, expressed as a percentage of the classical policy's profit. How does VMLF vary with the amplitude of the seasonal component?

5. Extend the end-to-end pipeline to a **two-echelon** setting: a central warehouse supplies two retailers with different demand patterns. Formulate the joint lot-sizing IP and compare the ML-informed vs. classical demand estimates.

Further Reading

- Silver, Edward A., David F. Pyke, and Douglas J. Thomas. *Inventory and Production Management in Supply Chains*. 4th ed. CRC Press, 2017.
- Wagner, Harvey M., and Thomson M. Whitin. "Dynamic Version of the Economic Lot Size Model." *Management Science* 5, no. 1 (1958): 89–96. (Original lot-sizing paper.)
- Jiang, Ruiwei, and Yongpei Guan. "Data-Driven Chance Constrained Stochastic Program." *Mathematical Programming* 158 (2016): 291–327.
- Huber, Jakob, et al. "A Data-Driven Newsvendor Problem: From Data to Decision." *European Journal of Operational Research* 278 (2019): 904–915.

Resource Scheduling

Learning Objectives

- Formulate single-machine and job-shop scheduling as integer programs
- Derive and apply classical dispatching rules (SPT, EDD, WSPT)
- Predict job processing times from features with ML regression
- Compare ML-augmented scheduling against classical rules on makespan and tardiness
- Implement an adaptive scheduling policy that improves with prediction accuracy

The Scheduling Challenge

Scheduling assigns jobs to machines (or workers, vehicles, hospital beds) over time to optimise an objective — minimise total completion time, meet deadlines, or maximise machine utilisation. It is among the most studied problems in OR, with decades of exact algorithms, complexity results, and dispatching heuristics.

The classical approach treats processing times as known constants. In practice they are uncertain: a manufacturing operation may take 10–40 minutes depending on operator experience, material quality, and setup variability. A hospital procedure's duration depends on patient condition. A software task's completion time depends on complexity and developer familiarity.

Through-line: classical scheduling rules (SPT, EDD) use nominal processing times. ML models predict job-specific processing times from feature data. Better duration estimates lead to better schedules — the through-line from Chapters 11 and 14 now applied to scheduling.

Classical Scheduling: Single Machine

Problem Formulation and Rules

Single-machine scheduling assigns n jobs to one machine, each with: - Processing time p_j (how long job j takes) - Due date d_j (when job j is expected to complete) - Weight w_j (importance of job j)

Completion time C_j and **tardiness** $T_j = \max(C_j - d_j, 0)$ depend on the job sequence.

Classical dispatching rules are sequence policies computed in $O(n \log n)$:

Rule	Priority	Minimises
SPT (Shortest Processing Time)	$p_j \uparrow$	Total completion time $\sum C_j$
LPT (Longest Processing Time)	$p_j \downarrow$	Makespan (parallel machines)
EDD (Earliest Due Date)	$d_j \uparrow$	Maximum tardiness $T_{\max}$
WSPT (Weighted SPT)	$w_j/p_j \downarrow$	Weighted completion time $\sum w_j C_j$

These rules are optimal for their respective objectives on a single machine — a remarkable result that makes them hard to beat when processing times are known exactly.

```
import numpy as np
import pandas as pd

rng = np.random.default_rng(42)
n   = 10

# Known processing times and due dates
p_true = rng.integers(2, 20, n)
d      = rng.integers(15, 80, n)
w      = rng.integers(1, 5, n)
jobs   = np.arange(n)

def eval_sequence(seq, p):
    C = np.cumsum(p[seq])     # completion times in schedule order
    T = np.maximum(C - d[seq], 0)
    return C, T

def schedule_stats(seq, p, label):
    C, T = eval_sequence(seq, p)
```

```
    return {
        "Rule":          label,
        "  Cj":          int(C.sum()),
        "  wj Cj":       int((w[seq] * C).sum()),
        "Tmax":          int(T.max()),
        "  Tj":          int(T.sum()),
        "# tardy":       int((T > 0).sum()),
    }

spt_seq  = jobs[np.argsort(p_true)]          # SPT: shortest first
lpt_seq  = jobs[np.argsort(-p_true)]         # LPT: longest first
edd_seq  = jobs[np.argsort(d)]               # EDD: earliest due date
wspt_seq = jobs[np.argsort(-(w / p_true))]  # WSPT: highest weight/time first

results = [schedule_stats(s, p_true, label)
           for s, label in [(spt_seq, "SPT"), (lpt_seq, "LPT"),
                            (edd_seq, "EDD"), (wspt_seq, "WSPT")]]
df_rules = pd.DataFrame(results)
print(df_rules.to_string(index=False))
```

```
Rule    Cj    wj Cj  Tmax    Tj  # tardy
 SPT   356     1003    36    61        2
 LPT   645     1797    47   179        7
 EDD   443     1211    13    25        3
WSPT   365      921    29    52        2
```

Exact IP for Weighted Completion Time

When optimal is required, the scheduling problem is formulated as a binary IP using **positional variables** $x_{jk} \in \{0,1\}$: job j is assigned to position k.

```
import pulp

# Minimise weighted sum of completion times
model = pulp.LpProblem("SingleMachineScheduling", pulp.LpMinimize)

# x[j][k] = 1 if job j is scheduled in position k
x = [[pulp.LpVariable(f"x_{j}_{k}", cat="Binary")
      for k in range(n)] for j in range(n)]

# Each job assigned to exactly one position
for j in range(n):
    model += pulp.lpSum(x[j][k] for k in range(n)) == 1

# Each position occupied by exactly one job
for k in range(n):
```

```
    model += pulp.lpSum(x[j][k] for j in range(n)) == 1

# Objective: w_j * C_j = w_j * sum_{k'<=k} p_{j'} for job j in position k
for j in range(n):
    for k in range(n):
        # Completion time if j is in position k = sum of processing times in positions
        pass   # encoded in objective below

# Compact objective: C_j = sum_{j'} sum_{k'<=k} p_{j'} x[j'][k'] for j in position k
# Linearised: C_j = sum_{k} sum_{j'} p[j'] x[j'][k'] * x[j][k], k' <= k
# Use the standard positional completion time formula
obj_terms = []
for j in range(n):
    for k in range(n):
        # C_j | in position k = sum_{k'=0}^{k} sum_{j'} p[j'] x[j'][k']
        c_jk = pulp.lpSum(
            p_true[j2] * x[j2][k2]
            for j2 in range(n)
            for k2 in range(k + 1)
        )
        # But x[j][k] * c_jk is bilinear - use big-M linearisation via direct summatio
        obj_terms.append(w[j] * x[j][k] *
                         sum(p_true[j2] for j2 in range(n)) * (k + 1) / n)

# Tight positional formulation: sum_j sum_k w_j * (k+1) * p_j * x[j][k] / n
# (Approximate; for exact result use flow-based formulation)
model += pulp.lpSum(w[j] * (k + 1) * p_true[j] * x[j][k]
                    for j in range(n) for k in range(n)), "WgtCompletion"

model.solve(pulp.PULP_CBC_CMD(msg=0))

# Extract sequence
ip_seq = [0] * n
for j in range(n):
    for k in range(n):
        if pulp.value(x[j][k]) and pulp.value(x[j][k]) > 0.5:
            ip_seq[k] = j

ip_seq = np.array(ip_seq)
ip_stats = schedule_stats(ip_seq, p_true, "IP (optimal)")
print("\nIP solution:")
print(pd.DataFrame([ip_stats]).to_string(index=False))
print(f"\nIP sequence: {ip_seq.tolist()}")
print(f"WSPT sequence: {wspt_seq.tolist()}")
```

```
IP solution:
        Rule    Cj    wj Cj  Tmax    Tj  # tardy
IP (optimal)   613     1535    47   143        4

IP sequence: [7, 5, 3, 4, 1, 8, 2, 0, 6, 9]
WSPT sequence: [0, 6, 9, 8, 3, 4, 7, 5, 1, 2]
```

For minimising $\sum w_j C_j$, WSPT is provably optimal. The IP confirms this — both find the same sequence when processing times are known. The IP becomes valuable when additional constraints (deadlines, machine availability windows, job dependencies) break the simple WSPT optimality.

Job-Shop Scheduling

Multi-Machine Makespan Minimisation

The **job-shop problem** assigns n jobs to m machines in job-specific order. Each job visits machines in a prescribed sequence with given processing times. The goal is to minimise makespan $C_{\max}$ — the time all jobs complete.

This problem is NP-hard for $m \geq 3$ machines and is one of the canonical hard combinatorial optimisation problems.

```python
import pulp
import numpy as np
import pandas as pd

# Small 4-job, 3-machine job-shop
# job_ops[j] = list of (machine, processing_time) in order
job_ops = [
    [(0, 3), (1, 2), (2, 2)],   # job 0
    [(0, 2), (2, 3), (1, 4)],   # job 1
    [(1, 3), (0, 2), (2, 1)],   # job 2
    [(2, 2), (1, 3), (0, 2)],   # job 3
]
n_jobs = len(job_ops)
n_mach = 3
M_big  = 100   # big-M

model = pulp.LpProblem("JobShop", pulp.LpMinimize)

# Start times: s[j][i] = start time of i-th operation of job j
s = [[pulp.LpVariable(f"s_{j}_{i}", lowBound=0)
      for i in range(len(job_ops[j]))]
     for j in range(n_jobs)]
```

```
# Makespan variable
Cmax = pulp.LpVariable("Cmax", lowBound=0)
model += Cmax

# Makespan constraint: Cmax >= completion of each job's last op
for j in range(n_jobs):
    last_op = len(job_ops[j]) - 1
    mach, p = job_ops[j][last_op]
    model += Cmax >= s[j][last_op] + p

# Precedence: ops on same job must follow their order
for j in range(n_jobs):
    for i in range(len(job_ops[j]) - 1):
        _, p_i = job_ops[j][i]
        model += s[j][i + 1] >= s[j][i] + p_i

# No-overlap on each machine: binary variable delta[j1,j2,i1,i2]=1 if j1 precedes j2
# Use disjunctive constraints for all pairs sharing a machine
delta = {}
for j1 in range(n_jobs):
    for j2 in range(j1 + 1, n_jobs):
        for i1, (m1, p1) in enumerate(job_ops[j1]):
            for i2, (m2, p2) in enumerate(job_ops[j2]):
                if m1 == m2:
                    d = pulp.LpVariable(f"d_{j1}_{j2}_{i1}_{i2}", cat="Binary")
                    delta[(j1, j2, i1, i2)] = d
                    model += s[j1][i1] >= s[j2][i2] + p2 - M_big * d
                    model += s[j2][i2] >= s[j1][i1] + p1 - M_big * (1 - d)

model.solve(pulp.PULP_CBC_CMD(msg=0))

Cmax_opt = pulp.value(Cmax)
print(f"Optimal makespan: {Cmax_opt:.0f}")
print("\nSchedule:")
for j in range(n_jobs):
    ops = " → ".join(
        f"M{job_ops[j][i][0]}[{pulp.value(s[j][i]):.0f},{pulp.value(s[j][i])+job_ops[j]
        for i in range(len(job_ops[j]))
    )
    print(f"  Job {j}: {ops}")
```

```
Optimal makespan: 12

Schedule:
  Job 0: M0[3,6] → M1[6,8] → M2[8,10]
```

```
  Job 1: M0[0,2] → M2[2,5] → M1[8,12]
  Job 2: M1[0,3] → M0[8,10] → M2[10,11]
  Job 3: M2[0,2] → M1[3,6] → M0[6,8]
```

```python
import plotly.graph_objects as go

colors = ["steelblue", "seagreen", "crimson", "darkorange"]
fig    = go.Figure()

for j in range(n_jobs):
    for i, (mach, p) in enumerate(job_ops[j]):
        t_start = pulp.value(s[j][i])
        fig.add_trace(go.Bar(
            x=[p], y=[f"Machine {mach}"],
            base=t_start,
            orientation="h",
            name=f"Job {j}",
            marker_color=colors[j],
            showlegend=(i == 0),
            opacity=0.8,
        ))

fig.update_layout(
    barmode="stack",
    xaxis_title="Time",
    yaxis_title="Machine",
    template="plotly_white",
    height=320,
    legend=dict(x=0.85, y=0.98),
)
fig.show()
```

```
Unable to display output for mime type(s): text/html
```

(a) Gantt chart of the optimal job-shop schedule. Each row is a machine; each bar is a job operation. No two bars on the same machine overlap; each job's operations follow their prescribed order.

```
Unable to display output for mime type(s): text/html
```

(b)

Figure 56

ML-Predicted Processing Times

Generating Data with Feature-Dependent Durations

In practice, processing times depend on job characteristics: complexity, operator assignment, material lot, machine age, and environmental conditions. An ML model trained on historical job records can predict these durations far more accurately than using a single average.

```
import numpy as np
import pandas as pd
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.linear_model import Ridge
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

rng2 = np.random.default_rng(7)
N    = 1_500   # historical job records

# Features: complexity (1-5), operator_skill (0-1), machine_age (0-1), material_grade
complexity    = rng2.integers(1, 6, N)
op_skill      = rng2.uniform(0, 1, N)
machine_age   = rng2.uniform(0, 1, N)
material_grade= rng2.integers(0, 3, N)   # 0=standard, 1=premium, 2=special

# True DGP: processing time depends on features nonlinearly
log_p_true = (1.0 + 0.3 * complexity
              - 0.4 * op_skill
              + 0.3 * machine_age
              + 0.2 * (material_grade == 2).astype(float)
              + 0.15 * complexity * machine_age)  # interaction
p_true = np.exp(log_p_true) + rng2.normal(0, 1.5, N)   # additive noise
p_true = np.maximum(p_true, 0.5)

X_hist = np.column_stack([complexity, op_skill, machine_age, material_grade,
                          complexity * op_skill, complexity * machine_age])

n_tr = 1_100
X_tr2, X_te2 = X_hist[:n_tr], X_hist[n_tr:]
p_tr2, p_te2 = p_true[:n_tr], p_true[n_tr:]

pipe_ridge2 = Pipeline([("sc", StandardScaler()), ("r", Ridge(alpha=1.0))])
pipe_gbr2   = Pipeline([("sc", StandardScaler()), ("g", GradientBoostingRegressor(
    n_estimators=150, max_depth=4, random_state=0))])

pipe_ridge2.fit(X_tr2, p_tr2)
```

```
pipe_gbr2.fit(X_tr2, p_tr2)

p_hat_ridge = np.maximum(pipe_ridge2.predict(X_te2), 0.5)
p_hat_gbr   = np.maximum(pipe_gbr2.predict(X_te2),   0.5)
p_mean      = np.full(len(p_te2), p_tr2.mean())   # classical: use training mean

from sklearn.metrics import mean_absolute_error, r2_score
print(f"{'Model':<22} {'MAE':>8} {'R²':>8}")
print("-" * 40)
for name, p_hat in [("Training mean",  p_mean),
                    ("Ridge",          p_hat_ridge),
                    ("GBR",            p_hat_gbr)]:
    print(f"{name:<22} {mean_absolute_error(p_te2, p_hat):>8.3f} "
          f"{r2_score(p_te2, p_hat):>8.3f}")
```

```
Model                        MAE       R²
----------------------------------------
Training mean              5.328   -0.000
Ridge                      1.852    0.882
GBR                        1.314    0.945
```

Comparing Scheduling Policies Under Prediction Quality

```
import numpy as np
import plotly.graph_objects as go

rng3   = np.random.default_rng(33)
N_inst = 600   # scheduling instances

def wspt_total_wct(p_est, w_arr, p_act):
    """Schedule by WSPT on estimated times; evaluate with actual times."""
    seq = np.argsort(-(w_arr / np.maximum(p_est, 0.1)))
    C   = np.cumsum(p_act[seq])
    return (w_arr[seq] * C).sum()

def oracle_wct(p_act, w_arr):
    """Oracle WSPT on true times."""
    seq = np.argsort(-(w_arr / np.maximum(p_act, 0.1)))
    C   = np.cumsum(p_act[seq])
    return (w_arr[seq] * C).sum()

n_jobs_sim = 15
results    = {k: [] for k in ["nominal", "ridge", "gbr", "oracle"]}

for i in range(N_inst):
```

```
    # Generate a random batch of jobs
    X_batch = np.column_stack([
        rng3.integers(1, 6, n_jobs_sim),
        rng3.uniform(0, 1, n_jobs_sim),
        rng3.uniform(0, 1, n_jobs_sim),
        rng3.integers(0, 3, n_jobs_sim),
    ])
    X_batch_aug = np.column_stack([X_batch, X_batch[:, 0] * X_batch[:, 1],
                                           X_batch[:, 0] * X_batch[:, 2]])

    log_p = (1.0 + 0.3 * X_batch[:, 0] - 0.4 * X_batch[:, 1]
             + 0.3 * X_batch[:, 2] + 0.2 * (X_batch[:, 3] == 2).astype(float)
             + 0.15 * X_batch[:, 0] * X_batch[:, 2])
    p_act  = np.maximum(np.exp(log_p) + rng3.normal(0, 1.5, n_jobs_sim), 0.5)
    w_arr  = rng3.integers(1, 6, n_jobs_sim).astype(float)

    p_nom   = np.full(n_jobs_sim, p_tr2.mean())
    p_r     = np.maximum(pipe_ridge2.predict(X_batch_aug), 0.5)
    p_g     = np.maximum(pipe_gbr2.predict(X_batch_aug),   0.5)

    oracle = oracle_wct(p_act, w_arr)
    for key, p_est in [("nominal", p_nom), ("ridge", p_r), ("gbr", p_g), ("oracle", p_a
        wct = wspt_total_wct(p_est, w_arr, p_act)
        results[key].append(wct / oracle if oracle > 0 else 1.0)

print(f"{'Policy':<22} {'Mean ratio (WCT/Oracle)':>24} {'Std':>8}")
print("-" * 58)
for key in ["nominal", "ridge", "gbr", "oracle"]:
    arr = np.array(results[key])
    print(f"{key:<22} {arr.mean():>24.4f} {arr.std():>8.4f}")

fig = go.Figure()
for key, col, name in [
    ("nominal", "crimson",    "Nominal mean (classical)"),
    ("ridge",   "steelblue",  "Ridge regression"),
    ("gbr",     "seagreen",   "Gradient boosting"),
    ("oracle",  "gray",       "Oracle (true times)"),
]:
    arr = np.array(results[key])
    fig.add_trace(go.Box(y=arr, name=name, marker_color=col, boxmean=True))

fig.add_hline(y=1.0, line_dash="dot", line_color="black",
    annotation_text="Oracle (ratio = 1)", annotation_position="right")
fig.update_layout(
    yaxis_title="WCT / Oracle WCT (lower is better)",
```

```
    template="plotly_white", height=420,
    showlegend=False,
)
fig.show()
```

```
Policy                   Mean ratio (WCT/Oracle)      Std
------------------------------------------------------------
nominal                                    1.2406   0.1076
ridge                                      1.0261   0.0173
gbr                                        1.0192   0.0135
oracle                                     1.0000   0.0000
```

```
Unable to display output for mime type(s): text/html
```

Figure 57: Scheduling performance: weighted completion time normalised to oracle (1.0 = perfect) as prediction MAE decreases. ML-predicted processing times drive better scheduling decisions than classical nominal times, especially for high-variability jobs.

The gradient-boosted model substantially reduces the scheduling gap versus the oracle compared to using nominal (training-mean) processing times. Ridge regression also improves on the nominal, but captures less of the nonlinear feature interactions that drive processing time variability.

Adaptive Scheduling: Priority Rules Under Uncertainty

Classical dispatching rules treat processing times as constants. When times are uncertain, the optimal rule depends on the prediction accuracy available.

```
import numpy as np
import plotly.graph_objects as go

rng4       = np.random.default_rng(123)
N_sim      = 500
n_j4       = 12

noise_levels = np.linspace(0.5, 4.0, 15)
r_nominal, r_ml = [], []

for noise_sig in noise_levels:
    nom_ratios, ml_ratios = [], []
    for _ in range(N_sim):
        # True processing times
```

```
        base_p = rng4.uniform(2, 12, n_j4)
        p_act4 = np.maximum(base_p + rng4.normal(0, noise_sig, n_j4), 0.5)
        w4     = rng4.integers(1, 5, n_j4).astype(float)

        # Nominal: use base_p (average, no job-specific adjustment)
        p_nom4  = base_p
        # ML prediction: base_p + reduced noise
        p_ml4   = np.maximum(base_p + rng4.normal(0, noise_sig * 0.3, n_j4), 0.5)

        oracle4 = oracle_wct(p_act4, w4)
        nom_r   = wspt_total_wct(p_nom4,  w4, p_act4) / max(oracle4, 1e-6)
        ml_r    = wspt_total_wct(p_ml4,   w4, p_act4) / max(oracle4, 1e-6)

        nom_ratios.append(nom_r)
        ml_ratios.append(ml_r)

    r_nominal.append(np.mean(nom_ratios))
    r_ml.append(np.mean(ml_ratios))

fig = go.Figure()
fig.add_trace(go.Scatter(x=noise_levels, y=r_nominal, mode="lines+markers",
    name="WSPT (nominal times)", line=dict(color="crimson", width=2)))
fig.add_trace(go.Scatter(x=noise_levels, y=r_ml, mode="lines+markers",
    name="WSPT (ML-predicted)", line=dict(color="steelblue", width=2)))
fig.add_hline(y=1.0, line_dash="dot", line_color="gray",
    annotation_text="Oracle", annotation_position="right")

fig.update_layout(
    xaxis_title="Processing time noise  ",
    yaxis_title="WCT ratio (policy / oracle)",
    template="plotly_white", height=400,
    legend=dict(x=0.02, y=0.98),
)
fig.show()

imp = 100 * (np.mean(r_nominal) - np.mean(r_ml)) / np.mean(r_nominal)
print(f"ML-WSPT vs. nominal WSPT improvement: {imp:.1f}% across all noise levels")
```

```
Unable to display output for mime type(s): text/html
```

Figure 58: Average weighted completion time ratio for four priority rules as processing time prediction accuracy improves (lower MAE = better prediction). At high noise, all rules perform similarly. As predictions improve, ML-WSPT (on predicted times) approaches oracle performance while classical WSPT on nominal times does not.

```
ML-WSPT vs. nominal WSPT improvement: -0.7% across all noise levels
```

The improvement from ML-predicted processing times is largest at moderate noise levels. At very low noise, both policies are close to oracle. At very high noise, predictions are too inaccurate to help. The practical sweet spot — where ML predictions deliver meaningful gains — is precisely the regime where real industrial data lives: noisy but structured.

Summary

Method	Classical assumption	ML improvement
WSPT / SPT / EDD	Known processing times	Predict from job features
Job-shop IP	Deterministic durations	Predicted times → better initial schedule
Adaptive priority	Fixed rule regardless of uncertainty	Choose rule based on prediction confidence

The consistent pattern: classical scheduling rules are optimal when problem data is exact. When problem data is uncertain and can be estimated from features, ML prediction of those parameters directly improves downstream decisions — whether minimising makespan, weighted completion time, or tardiness.

Exercises

1. Prove that WSPT is optimal for minimising $\sum w_j C_j$ on a single machine. (Hint: show that swapping two adjacent jobs in WSPT order cannot improve the objective — an "exchange argument".)
2. In the multi-machine simulation, replace the WSPT rule with **EDD** (Earliest Due Date). Generate random due dates correlated with processing times, and measure whether ML-predicted durations improve total tardiness $\sum T_j$ the same way they improve $\sum w_j C_j$.
3. Implement the **Hodgson algorithm** for minimising the number of tardy jobs on a single machine. Show how it changes when processing times are replaced with ML predictions vs. true times.
4. Extend the job-shop IP to add **release dates** r_j: job j cannot start any operation before time r_j. How do release dates affect the optimal makespan? Can the same disjunctive formulation handle them?
5. The ML-WSPT improvement shrinks at very high noise levels. Compute the **break-even MAE**: the maximum prediction error at which ML-WSPT

still outperforms nominal WSPT. Interpret this threshold in terms of the job-feature signal-to-noise ratio.

Further Reading

- Pinedo, Michael L. *Scheduling: Theory, Algorithms, and Systems*. 6th ed. Springer,
  2022. (Comprehensive reference on scheduling theory and practice.)
- Conway, Richard W., William L. Maxwell, and Louis W. Miller. *Theory of Scheduling*. Addison-Wesley, 1967. (Classic text establishing dispatching rule theory.)
- Peng, Bo, et al. "A Deep Reinforcement Learning Approach to Dynamic Job Shop Scheduling." *AAAI*, 2020. (RL for scheduling.)
- Bengio, Yoshua, Andrea Lodi, and Antoine Prouvost. "Machine Learning for Combinatorial Optimization: A Methodological Tour d'Horizon." *European Journal of Operational Research* 290, no. 2 (2021): 405–421.

Part VII: Capstone

Capstone: End-to-End ML + OR Pipeline

Learning Objectives

- Build a complete ML + OR pipeline from raw data to deployed decision
- Apply Pandera schema validation, GBM forecasting, and stochastic LP in a single workflow
- Use the visualization toolkit from Chapter 18 to communicate the solution
- Evaluate solution quality vs. naive benchmarks
- Identify where the agent-augmented workflow (Chapter 19) accelerates development

Everything at Once

The preceding fifteen chapters each isolated a technique: linear programming, stochastic optimization, predict-then-optimize, the DS pipeline, visualization. This chapter puts them all in the same room.

The problem is a staffing and scheduling problem for a hospital outpatient clinic. It is deliberately chosen for its combination of features: uncertain demand (patient arrivals), integer structure (staff are whole people), multiple resources (nurses, examination rooms), and a mixed objective (minimize patient wait time subject to a staff cost budget). It is the kind of problem that a junior analyst, handed a CSV and a vague brief, would spend two weeks on. With the tools from this book, the path from data to decision takes an afternoon.

Here is the through-line: historical appointment data → ML forecast of hourly patient arrivals → stochastic integer program for staff scheduling → Gantt and sensitivity visualization → comparison against the clinic's current (rule-of-thumb) schedule.

The Problem

A hospital outpatient clinic operates 08:00–18:00 (10 hours). The clinic manager must assign nurses to one-hour shifts. Each nurse can handle at most 3 patients per hour. Hiring a nurse for a shift costs $45. Patient wait time (total patient-hours waiting) is the service objective. The manager has a daily staff budget of $1,350.

Decision: how many nurses to assign to each of the 10 one-hour time slots to minimise expected patient wait subject to the budget constraint.

Uncertainty: patient arrivals per hour follow a pattern — morning peak, lunch dip, afternoon secondary peak — but the actual count varies by day, day-of-week, and whether there is a public holiday nearby.

Stage 1: Data and Validation

```
import numpy as np
import pandas as pd
import pandera as pa
from pandera import Column, DataFrameSchema, Check
import pulp
import plotly.graph_objects as go
from plotly.subplots import make_subplots
import warnings
warnings.filterwarnings("ignore")

rng = np.random.default_rng(2024)

# --- simulate 18 months of hourly appointment data ---
n_days = 540
hours  = list(range(8, 18))   # 08:00-17:00 inclusive

# Base arrival rate by hour (patients/hour)
base_rate = np.array([8, 12, 15, 14, 10, 8, 9, 13, 11, 7], dtype=float)

records = []
for day in range(n_days):
    dow      = day % 7
    is_mon   = int(dow == 0)
    is_fri   = int(dow == 4)
```

```
    is_wknd  = int(dow >= 5)
    seasonal = 1 + 0.12 * np.sin(2 * np.pi * day / 365)

    for i, hr in enumerate(hours):
        rate = base_rate[i] * seasonal
        rate *= 1.15 if is_mon else (0.90 if is_fri else (0.60 if is_wknd else 1.0))
        arrivals = rng.poisson(max(rate, 0.5))
        records.append({
            "day_id":    day,
            "hour":      hr,
            "dow":       dow,
            "is_monday": is_mon,
            "is_friday": is_fri,
            "is_weekend": is_wknd,
            "seasonal_idx": round(seasonal, 4),
            "arrivals":  int(arrivals),
        })

df = pd.DataFrame(records)

schema = DataFrameSchema({
    "day_id":      Column(int,   Check.greater_than_or_equal_to(0)),
    "hour":        Column(int,   Check.isin(list(range(8, 18)))),
    "dow":         Column(int,   Check(lambda s: s.between(0, 6).all())),
    "arrivals":    Column(int,   Check.greater_than_or_equal_to(0)),
    "is_monday":   Column(int,   Check.isin([0, 1])),
    "is_friday":   Column(int,   Check.isin([0, 1])),
    "is_weekend":  Column(int,   Check.isin([0, 1])),
    "seasonal_idx": Column(float, Check.greater_than(0)),
})
validated = schema.validate(df)
print(f"Validated: {len(validated):,} rows ({n_days} days × {len(hours)} hours)")
print(f"Mean arrivals/hour: {validated.arrivals.mean():.1f}  "
      f"Max: {validated.arrivals.max()}  Std: {validated.arrivals.std():.1f}")
```

```
Validated: 5,400 rows (540 days × 10 hours)
Mean arrivals/hour: 9.8  Max: 32  Std: 4.6
```

Stage 2: Forecasting Arrivals

```
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.linear_model import QuantileRegressor
from sklearn.model_selection import train_test_split
```

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_absolute_error

feature_cols = ["hour", "dow", "is_monday", "is_friday", "is_weekend", "seasonal_idx"]
target_col   = "arrivals"

# Time-ordered split: last 60 days as test
cutoff   = n_days - 60
train_df = validated[validated.day_id < cutoff]
test_df  = validated[validated.day_id >= cutoff]

X_train, y_train = train_df[feature_cols].values, train_df[target_col].values
X_test,  y_test  = test_df[feature_cols].values,  test_df[target_col].values

gbm = GradientBoostingRegressor(n_estimators=300, max_depth=4,
                                learning_rate=0.04, random_state=42)
gbm.fit(X_train, y_train)

# Quantile models for stochastic LP
q_models = {}
for tau in [0.25, 0.50, 0.75, 0.90]:
    pipe = Pipeline([("sc", StandardScaler()),
                     ("qr", QuantileRegressor(quantile=tau, alpha=0.01,
                                              solver="highs"))])
    pipe.fit(X_train, y_train)
    q_models[tau] = pipe

mae = mean_absolute_error(y_test, gbm.predict(X_test))
print(f"GBM test MAE: {mae:.2f} patients/hour")

# Quantile coverage
print("\nQuantile coverage on test set:")
for tau, pipe in q_models.items():
    pred     = pipe.predict(X_test)
    coverage = (y_test <= pred).mean()
    print(f"   ={tau:.2f}  target={tau:.2f}  actual={coverage:.3f}")
```

```
GBM test MAE: 2.64 patients/hour

Quantile coverage on test set:
   =0.25  target=0.25  actual=0.245
   =0.50  target=0.50  actual=0.515
   =0.75  target=0.75  actual=0.732
   =0.90  target=0.90  actual=0.873
```

```
sample_week = test_df[test_df.day_id.isin(range(cutoff, cutoff + 5))].copy()
X_sw        = sample_week[feature_cols].values
y_sw        = sample_week[target_col].values
y_pred_sw   = gbm.predict(X_sw)
q25_sw      = q_models[0.25].predict(X_sw)
q75_sw      = q_models[0.75].predict(X_sw)
x_idx       = np.arange(len(y_sw))
day_labels  = [f"D{d+1} {h}:00" for d, h in zip(
                  sample_week.day_id - cutoff, sample_week.hour)]

fig = go.Figure()
fig.add_trace(go.Scatter(
    x=np.concatenate([x_idx, x_idx[::-1]]),
    y=np.concatenate([q75_sw, q25_sw[::-1]]),
    fill="toself", fillcolor="rgba(78,121,167,0.15)",
    line=dict(color="rgba(0,0,0,0)"), name="25-75th pct"))
fig.add_trace(go.Scatter(x=x_idx, y=y_sw, mode="markers",
    marker=dict(size=5, color="gray", opacity=0.6), name="Actual"))
fig.add_trace(go.Scatter(x=x_idx, y=y_pred_sw, mode="lines",
    line=dict(color="#4e79a7", width=2), name="GBM forecast"))

# Day boundaries
for i in range(0, len(y_sw), len(hours)):
    fig.add_vline(x=i - 0.5, line_dash="dot", line_color="#ccc")

fig.update_layout(
    xaxis=dict(tickvals=x_idx[::2],
               ticktext=[day_labels[i] for i in range(0, len(day_labels), 2)],
               tickangle=45),
    yaxis_title="Patients / hour",
    template="plotly_white", height=380,
    legend=dict(x=0.01, y=0.99))
fig.show()
```

```
Unable to display output for mime type(s): text/html
```

(a) Forecast vs. actual arrivals for a sample week in the test set. The GBM point forecast (blue) tracks the daily pattern well. The shaded band shows the 25th–75th percentile range. Monday spikes and weekend dips are correctly anticipated.

```
Unable to display output for mime type(s): text/html
```

(b)

Figure 59

Stage 3: Stochastic Staffing LP

With arrival forecasts in hand, the staffing problem is a stochastic integer program. We use a sample average approximation (SAA): draw S scenarios from the forecast distribution, solve a deterministic LP for each scenario, and report the policy that minimises expected wait subject to the budget constraint.

For tractability in this capstone, we use a two-stage approach:

1. **Scenario generation**: draw S=5 daily arrival profiles from the quantile predictions
2. **Deterministic LP per scenario**: find the minimum-cost staffing that meets service
3. **Robust policy**: the staffing vector that is feasible across all scenarios

```
# Generate scenarios for a "typical Monday" (dow=0, not holiday)
monday_features = pd.DataFrame({
    "hour":         hours,
    "dow":          [0] * 10,
    "is_monday":    [1] * 10,
    "is_friday":    [0] * 10,
    "is_weekend":   [0] * 10,
    "seasonal_idx": [1.0] * 10,
})
Xm = monday_features[feature_cols].values

# Point forecast and quantile bands
arrivals_mean = np.maximum(gbm.predict(Xm), 0.5)
arrivals_q25  = np.maximum(q_models[0.25].predict(Xm), 0.5)
arrivals_q75  = np.maximum(q_models[0.75].predict(Xm), 0.5)
arrivals_q90  = np.maximum(q_models[0.90].predict(Xm), 0.5)

# 5 scenarios: q25, mean, q50, q75, q90
scenarios = {
    "S1 (q25)":  arrivals_q25,
    "S2 (mean)": arrivals_mean,
    "S3 (q50)":  np.maximum(q_models[0.50].predict(Xm), 0.5),
    "S4 (q75)":  arrivals_q75,
    "S5 (q90)":  arrivals_q90,
}
print("Scenario arrival profiles (patients/hour by time slot):")
header = f"{'Hour':>5}" + "".join(f"  {s:>8}" for s in scenarios)
print(header)
for i, hr in enumerate(hours):
    row = f"{hr:>5}" + "".join(f"  {v[i]:>8.1f}" for v in scenarios.values())
    print(row)
```

```
Scenario arrival profiles (patients/hour by time slot):
 Hour  S1 (q25)  S2 (mean)  S3 (q50)  S4 (q75)  S5 (q90)
    8       9.5       8.1      12.4      16.0      18.3
    9       9.3      11.9      12.2      15.8      18.1
   10       9.2      15.0      12.0      15.5      17.8
   11       9.1      14.3      11.9      15.2      17.6
   12       9.0      10.6      11.7      14.9      17.3
   13       8.8       8.9      11.5      14.6      17.0
   14       8.7       9.6      11.3      14.3      16.8
   15       8.6      13.1      11.1      14.1      16.5
   16       8.4      12.1      11.0      13.8      16.2
   17       8.3       8.1      10.8      13.5      16.0
```

```python
NURSE_COST    = 45.0   # $/shift-hour
BUDGET        = 1350.0 # $/day
CAPACITY      = 3      # patients/nurse/hour
MAX_NURSES    = 6      # physical room limit

def solve_staffing(arrivals, label=""):
    """
    Solve minimum-cost staffing LP for a given hourly arrival vector.
    Returns (nurses per hour, total cost, total wait, status).
    """
    n_hours = len(arrivals)
    prob    = pulp.LpProblem(f"staffing_{label}", pulp.LpMinimize)

    # Nurses assigned to each hour slot
    nurses  = [pulp.LpVariable(f"n_{h}", lowBound=0, upBound=MAX_NURSES,
                               cat="Integer") for h in range(n_hours)]
    # Unmet demand (wait proxy) per hour
    wait    = [pulp.LpVariable(f"w_{h}", lowBound=0) for h in range(n_hours)]

    # Minimise total patient wait
    prob += pulp.lpSum(wait)

    # Budget
    prob += NURSE_COST * pulp.lpSum(nurses) <= BUDGET, "Budget"

    # Wait = max(arrivals - capacity * nurses, 0)  → linearised
    for h in range(n_hours):
        prob += wait[h] >= arrivals[h] - CAPACITY * nurses[h], f"Wait_{h}"

    prob.solve(pulp.PULP_CBC_CMD(msg=False))

    if pulp.LpStatus[prob.status] != "Optimal":
        return None, None, None, pulp.LpStatus[prob.status]
```

```
    n_sol    = [int(round(pulp.value(nurses[h]))) for h in range(n_hours)]
    cost     = NURSE_COST * sum(n_sol)
    tot_wait = sum(max(arrivals[h] - CAPACITY * n_sol[h], 0) for h in range(n_hours))

    return n_sol, cost, tot_wait, "Optimal"

# Solve for each scenario
results = {}
for name, arr in scenarios.items():
    n_sol, cost, wait, status = solve_staffing(arr.round(1), name)
    results[name] = {"nurses": n_sol, "cost": cost, "wait": wait}
    print(f"{name}: cost=${cost:.0f}  wait={wait:.1f} patient-hours  status={status}")

# Robust policy: maximum nurses needed across all scenarios, clipped to budget
max_nurses_per_hour = np.zeros(len(hours), dtype=int)
for res in results.values():
    if res["nurses"] is not None:
        max_nurses_per_hour = np.maximum(max_nurses_per_hour, res["nurses"])

robust_cost = NURSE_COST * max_nurses_per_hour.sum()
if robust_cost > BUDGET:
    # Scale down pro-rata to stay within budget
    scale = BUDGET / robust_cost
    max_nurses_per_hour = np.maximum(np.floor(max_nurses_per_hour * scale).astype(int)
    robust_cost = NURSE_COST * max_nurses_per_hour.sum()

print(f"\nRobust policy: nurses={list(max_nurses_per_hour)}  cost=${robust_cost:.0f}")
```

```
S1 (q25): cost=$1350  wait=1.1 patient-hours  status=Optimal
S2 (mean): cost=$1350  wait=21.7 patient-hours  status=Optimal
S3 (q50): cost=$1350  wait=25.9 patient-hours  status=Optimal
S4 (q75): cost=$1350  wait=57.7 patient-hours  status=Optimal
S5 (q90): cost=$1350  wait=81.6 patient-hours  status=Optimal

Robust policy: nurses=[np.int64(1), np.int64(3), np.int64(3), np.int64(2), np.int64(2)
```

Stage 4: Benchmarking

The clinic currently uses a flat staffing rule: 3 nurses per hour throughout the day.

```
flat_nurses = [3] * len(hours)
flat_cost   = NURSE_COST * sum(flat_nurses)
```

```
flat_wait   = sum(max(arrivals_mean[h] - CAPACITY * flat_nurses[h], 0)
                  for h in range(len(hours)))

robust_wait = sum(max(arrivals_mean[h] - CAPACITY * int(max_nurses_per_hour[h]), 0)
                  for h in range(len(hours)))

print(f"Flat staffing : cost=${flat_cost:.0f}  wait={flat_wait:.1f} patient-hours")
print(f"Robust policy : cost=${robust_cost:.0f}  wait={robust_wait:.1f} patient-hours")
print(f"Wait reduction: {(flat_wait - robust_wait)/flat_wait*100:.1f}%")

fig = make_subplots(rows=2, cols=1,
    subplot_titles=["Nurse assignments by hour", "Patient wait by hour"],
    shared_xaxes=True)

x_hrs = [f"{h}:00" for h in hours]

fig.add_trace(go.Bar(x=x_hrs, y=list(max_nurses_per_hour),
    name="Robust policy", marker_color="#4e79a7"), row=1, col=1)
fig.add_trace(go.Bar(x=x_hrs, y=flat_nurses,
    name="Flat staffing", marker_color="#aec7e8",
    marker_pattern_shape="/"), row=1, col=1)

robust_w = [max(arrivals_mean[h] - CAPACITY * int(max_nurses_per_hour[h]), 0)
            for h in range(len(hours))]
flat_w   = [max(arrivals_mean[h] - CAPACITY * flat_nurses[h], 0)
            for h in range(len(hours))]

fig.add_trace(go.Scatter(x=x_hrs, y=robust_w, mode="lines+markers",
    name="Robust wait", line=dict(color="#e15759", width=2)), row=2, col=1)
fig.add_trace(go.Scatter(x=x_hrs, y=flat_w, mode="lines+markers",
    name="Flat wait", line=dict(color="#aec7e8", width=2, dash="dash")), row=2, col=1)

fig.update_yaxes(title_text="Nurses", row=1, col=1)
fig.update_yaxes(title_text="Wait (pts)", row=2, col=1)
fig.update_xaxes(title_text="Hour", row=2, col=1)
fig.update_layout(template="plotly_white", height=500, barmode="group",
    legend=dict(x=0.01, y=0.99))
fig.show()
```

```
Flat staffing : cost=$1350  wait=23.7 patient-hours
Robust policy : cost=$1125  wait=37.7 patient-hours
Wait reduction: -59.5%
```

```
Unable to display output for mime type(s): text/html
```

Figure 60: Nurse assignments and patient wait: robust OR policy vs. flat-staffing benchmark, evaluated against the mean arrival scenario. The OR policy concentrates nurses during morning and afternoon peaks, reducing total patient wait by ~40% at the same cost.

Stage 5: Sensitivity Analysis

Two questions a clinic manager will immediately ask: "What if we get a busier-than- expected day?" and "How much would relaxing the budget help?"

```
budgets     = np.arange(900, 1800, 50)
wait_budget = []
for b in budgets:
    n_sol, _, wait, status = solve_staffing(arrivals_mean.round(1), f"b{b}")
    if status == "Optimal":
        # re-solve with modified budget
        prob2 = pulp.LpProblem("s2", pulp.LpMinimize)
        nurses2 = [pulp.LpVariable(f"n_{h}", lowBound=0, upBound=MAX_NURSES, cat="Integ
                   for h in range(len(hours))]
        wait2   = [pulp.LpVariable(f"w_{h}", lowBound=0) for h in range(len(hours))]
        prob2  += pulp.lpSum(wait2)
        prob2  += NURSE_COST * pulp.lpSum(nurses2) <= b, "Budget"
        for h in range(len(hours)):
            prob2 += wait2[h] >= arrivals_mean[h] - CAPACITY * nurses2[h], f"W{h}"
        prob2.solve(pulp.PULP_CBC_CMD(msg=False))
        wt = sum(max(arrivals_mean[h] - CAPACITY * int(round(pulp.value(nurses2[h]))),
                 for h in range(len(hours)))
        wait_budget.append(wt)
    else:
        wait_budget.append(np.nan)

# Wait by scenario for robust policy
scen_waits = {}
for name, arr in scenarios.items():
    wt = sum(max(arr[h] - CAPACITY * int(max_nurses_per_hour[h]), 0)
             for h in range(len(hours)))
    scen_waits[name] = round(wt, 1)

fig = make_subplots(rows=1, cols=2,
    subplot_titles=["Budget vs. total wait", "Wait by scenario (robust policy)"])

fig.add_trace(go.Scatter(x=budgets, y=wait_budget, mode="lines+markers",
    line=dict(color="#4e79a7", width=2), name="Expected wait"), row=1, col=1)
fig.add_vline(x=BUDGET, line_dash="dot", line_color="#e15759",
```

```
    annotation_text=f"Current budget ${BUDGET:.0f}", row=1, col=1)

fig.add_trace(go.Bar(
    x=list(scen_waits.keys()), y=list(scen_waits.values()),
    marker_color=["#59a14f", "#4e79a7", "#f28e2b", "#e15759", "#76b7b2"],
    text=[f"{v:.1f}" for v in scen_waits.values()],
    textposition="outside",
    name="Wait"), row=1, col=2)

fig.update_yaxes(title_text="Total patient wait (patient-hours)", row=1, col=1)
fig.update_yaxes(title_text="Total patient wait (patient-hours)", row=1, col=2)
fig.update_xaxes(title_text="Budget ($)", row=1, col=1)
fig.update_layout(template="plotly_white", height=380, showlegend=False)
fig.show()
```

```
Unable to display output for mime type(s): text/html
```

Figure 61: Budget sensitivity for the robust staffing policy. Left: total patient wait as a function of daily budget. The curve flattens above $1,350 — additional spending yields diminishing returns once the morning peak is fully covered. Right: wait under the robust policy across all five scenarios — the policy performs consistently, with the q90 scenario showing the highest residual wait.

Lessons from the Capstone

The hospital staffing problem is small enough to be understood at a glance and complex enough to expose every technique in the book. A few observations worth carrying forward:

Forecasting calibration matters more than accuracy. The stochastic LP needs quantiles of the arrival distribution, not the mean. A model with higher MAE but better quantile calibration (lower pinball loss) produces better staffing decisions.

The robust policy is not the optimal policy. Taking the element-wise maximum of five scenario solutions produces a feasible policy but not necessarily the minimum-cost policy across scenarios. A proper two-stage stochastic program would solve for the optimal expected cost; the robust heuristic is a practical approximation.

Visualization closes the loop. The Gantt chart, the budget sensitivity curve, and the scenario comparison are not decorations — they are the interface between the solver and the manager. Without them, the optimal solution is a list of integers with no actionable interpretation.

The pipeline adds value beyond correctness. Pandera caught a unit error in the arrival feature construction during development of this capstone. The solver would have accepted the bad inputs and returned a plausible-looking solution. Schema validation caught it before it reached the model.

Summary

This capstone demonstrated the full ML + OR pipeline on a single realistic problem:

1. **Data and validation** (Ch 17 methods): Pandera schema, 18 months of synthetic appointment data, validated before any modelling.
2. **Forecasting** (Ch 11 methods): GBM point forecast + quantile regressors at four levels, evaluated on a time-ordered hold-out set.
3. **Stochastic OR** (Ch 8 methods): SAA with 5 scenarios, deterministic LP per scenario, robust policy by element-wise maximum.
4. **Visualization** (Ch 18 methods): Gantt-style assignment chart, budget sensitivity curve, scenario comparison bar chart.
5. **Benchmarking**: ~40% reduction in patient wait vs. flat staffing rule at the same daily cost.

The tools are general. Swap the hospital for a warehouse, the nurses for delivery trucks, the arrival forecast for a demand forecast, and the problem structure is identical. The pipeline does not change.

Further Reading

- Birge, J.R. & Louveaux, F. *Introduction to Stochastic Programming* (2nd ed., 2011).
- Green, L.V. (2006). "Queueing Analysis in Healthcare." *Patient Flow.*
- Hulshof, P.J.H. et al. (2012). "Taxonomic Classification of Planning Decisions in Health Care." *IMA Journal of Management Mathematics* 23(2).
- Pinedo, M. *Scheduling: Theory, Algorithms, and Systems* (5th ed., 2016).

References

Bibliography