STAT 4830: Numerical optimization for data science and ML

Lecture 2: Linear Regression: Direct Methods

Professor Damek Davis

Overview

1. Introduction
2. Prediction with Multiple Features
3. Computing Predictions Efficiently
4. Finding Optimal Weights
5. Direct Solution Methods
6. Effect and Remedy for Numerical Instability
7. QR Factorization: A More Stable Approach
8. The Limits of Direct Methods: Scaling Up

Introduction

Last lecture: PyTorch's efficient handling of vectors and matrices

Today: Applying these tools to prediction - a core data science challenge

Four key steps:

1. Converting predictions into matrix operations
2. Formulating the optimization problem
3. Converting optimization into linear equations
4. Solving equations efficiently via direct methods

Prediction with Multiple Features

Basic house price equation:

$$\text{price} = w_1 \cdot \text{size} + w_2 \cdot \text{age} + w_3 \cdot \text{bedrooms} + w_4 \cdot \text{location} + \text{noise}$$

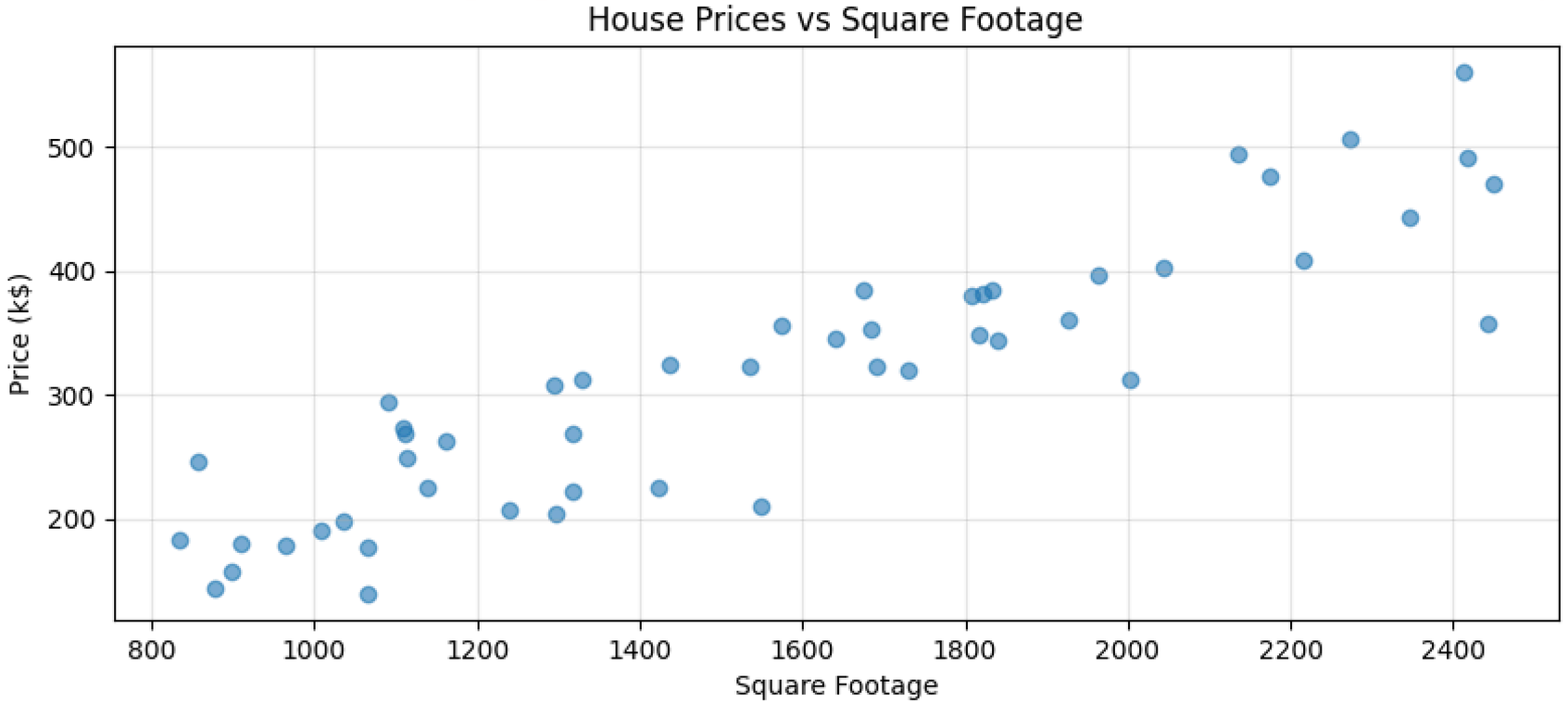
Vector notation:

$$y = w^T x + \epsilon$$

Each weight has clear meaning:

- w_1: dollars per square foot
- w_2: price change per year of age
- w_3: value per bedroom
- w_4: location premium

Feature Mapping and Error Analysis



Why linear models often work:

- Linear relationships in real data
- Scatter represents noise/unexplained factors
- Simple but powerful approximation

Code Example: House Price Prediction

```
house = {
    'size': 1500,      # x₁: sq ft
    'age': 10,         # x₂: years
    'bedrooms': 3,     # x₃: count
    'location': 0.8    # x₄: some score
}
price = 500000  # y: dollars

def predict_price(house, weights):
    """Predict house price using linear combination of features"""
    return (
        weights[0] * house['size'] +      # dollars per sq ft
        weights[1] * house['age'] +       # price change per year
        weights[2] * house['bedrooms'] +  # value per bedroom
        weights[3] * house['location']    # location premium
    )
```

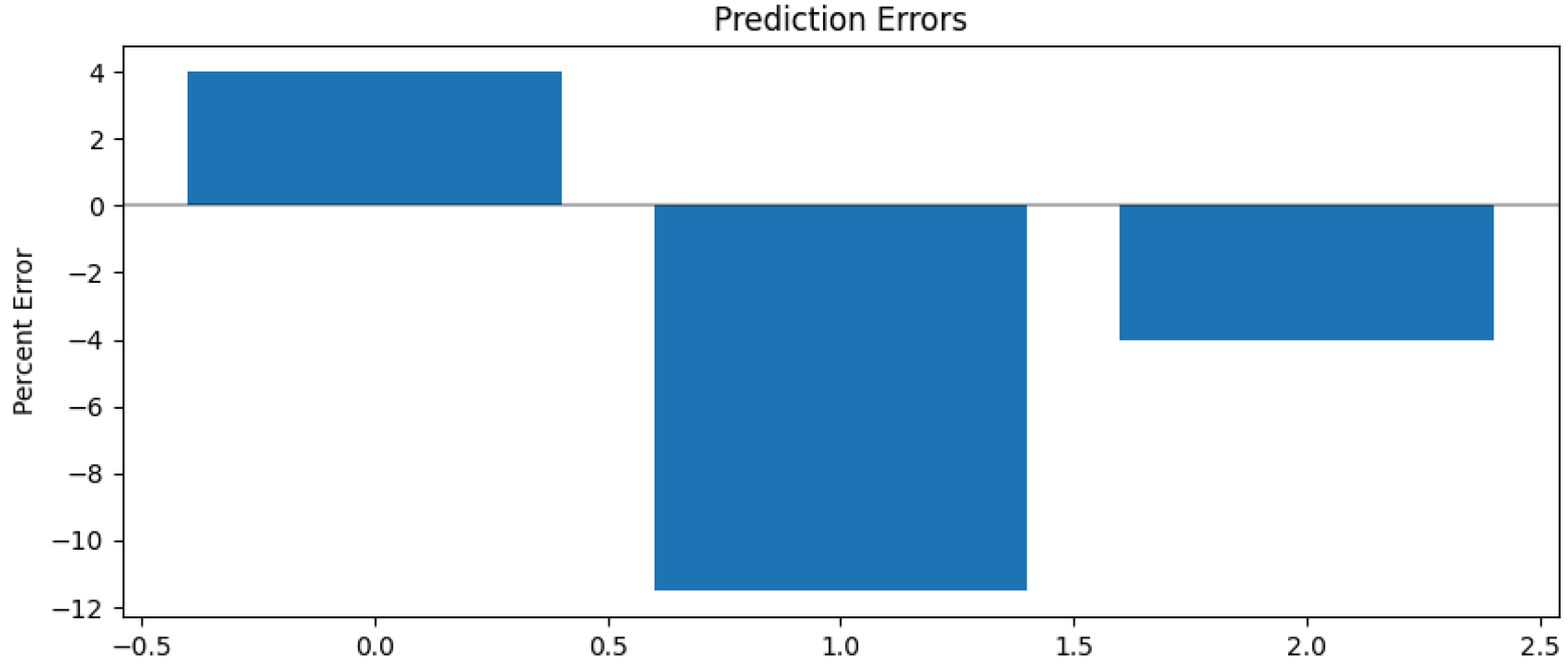
Testing Our Model

```
X = torch.tensor([
    [1500, 10, 3, 0.8],  # house 1
    [2100, 2,  4, 0.9],  # house 2
    [800,  50, 2, 0.3]   # house 3
], dtype=torch.float32)
y = torch.tensor([500000, 800000, 250000], dtype=torch.float32)
weights = torch.tensor([200, -1000, 50000, 100000], dtype=torch.float32)

predictions = X @ weights  # Matrix multiplication!

# Results:
# House 1: $520,000 (Error: +4.0%)
# House 2: $708,000 (Error: -11.5%)
# House 3: $240,000 (Error: -4.0%)
```

Prediction Errors



Our model's performance:

- Errors range from 4% to 12%
- Systematic patterns?
- Can we do better?

Computing Predictions Efficiently

Matrix multiplication for all predictions:

$$\text{house}_1 : [1500, 10, 3, 0.8] \cdot [w_1, w_2, w_3, w_4] = \text{prediction}_1$$

$$\text{house}_2 : [2100, 2, 4, 0.9] \cdot [w_1, w_2, w_3, w_4] = \text{prediction}_2$$

$$\text{house}_3 : [800, 50, 2, 0.3] \cdot [w_1, w_2, w_3, w_4] = \text{prediction}_3$$

Feature matrix:

$$X = \begin{bmatrix} \text{size}_1 & \text{age}_1 & \text{beds}_1 & \text{loc}_1 \\ \text{size}_2 & \text{age}_2 & \text{beds}_2 & \text{loc}_2 \\ \text{size}_3 & \text{age}_3 & \text{beds}_3 & \text{loc}_3 \end{bmatrix} = \begin{bmatrix} 1500 & 10 & 3 & 0.8 \\ 2100 & 2 & 4 & 0.9 \\ 800 & 50 & 2 & 0.3 \end{bmatrix}$$

Performance Impact

```
Dataset Size | Loop Time   | Matrix Time | Speedup
-------------------------------------------------
     1,000   |   0.21ms    |    0.01ms   |   21x
    10,000   |   1.79ms    |    0.05ms   |   34x
   100,000   |  19.39ms    |    0.58ms   |   33x
 1,000,000   | 196.33ms    |    5.43ms   |   36x
```

Why so fast?

- CPU's SIMD instructions
- Cache-friendly memory access
- Optimized BLAS libraries
- Critical for iterative methods later!

Finding Optimal Weights

Error function:

$$\text{error} = \sum_{i=1}^{n} (y_i - w^T x_i)^2$$

Partial derivatives:

$$\frac{\partial}{\partial w_j} \text{error} = -2 \sum_{i=1}^{n} x_{ij} (y_i - w^T x_i) = 0$$

Matrix form:

$$-2X^T(y - Xw) = 0$$

$$X^T X w = X^T y$$

Finding Optimal Weights: The Math in 1D

Remember our 10% error on house prices? Let's discover why calculus and linear algebra together give us a direct path to the best weights.

Simple example with two houses:

- House 1: 1000 sq ft → 300k dollars
- House 2: 2000 sq ft → 600k dollars

Notice: When size doubles (1000 → 2000), price doubles too (300k → 600k)

Finding Optimal Weights in 1D

Error for a given weight w (price per sq ft, in k):

$$\mathbf{error}(w) = (300 - 1000w)^2 + (600 - 2000w)^2$$

To minimize: Set derivative to zero and solve

$$-2(1000)(300 - 1000w) - 2(2000)(600 - 2000w) = 0$$

Collecting terms:

$$(1000^2 + 2000^2)w = 1000(300) + 2000(600)$$

In terms of data matrix:

$$X^T X w = X^T y$$

More Generally: Calculus → Lin Alg

For multiple features, we minimize:

$$\text{error} = \sum_{i=1}^{n} (y_i - w^T x_i)^2$$

Taking partial derivatives:

$$\frac{\partial}{\partial w_j} \text{error} = -2 \sum_{i=1}^{n} x_{ij} (y_i - w^T x_i) = 0$$

In matrix form:

$$-2X^T(y - Xw) = 0$$

$$X^T X w = X^T y$$

The Normal Equations

> Calculus turns "minimize prediction error" into "solve the **normal equations**"

$$X^T X w = X^T y$$

$$X^T X = \begin{bmatrix} \textbf{size} \cdot \textbf{size} & \textbf{size} \cdot \textbf{age} & \textbf{size} \cdot \textbf{beds} \\ \textbf{age} \cdot \textbf{size} & \textbf{age} \cdot \textbf{age} & \textbf{age} \cdot \textbf{beds} \\ \textbf{beds} \cdot \textbf{size} & \textbf{beds} \cdot \textbf{age} & \textbf{beds} \cdot \textbf{beds} \end{bmatrix}$$

These equations have beautiful properties:

1. One equation per weight
2. Linear in the weights
3. Error vector $(Xw_\star - y)$ at solution $w_\star$ becomes orthogonal to X

The Normal Equations

> Calculus turns minimizing prediction error into solving linear equations

$$X^T X w = X^T y$$

System size depends on features, not data:

- n houses, p features
- X is $n \times p$
- $X^T X$ is $p \times p$
- Even with millions of houses, system stays small!

Structure of Normal Equations

When we multiply $X^T X$, each entry combines feature vectors:

$$A = X^T X = \begin{bmatrix} \textbf{size} \cdot \textbf{size} & \textbf{size} \cdot \textbf{age} & \textbf{size} \cdot \textbf{beds} \\ \textbf{age} \cdot \textbf{size} & \textbf{age} \cdot \textbf{age} & \textbf{age} \cdot \textbf{beds} \\ \textbf{beds} \cdot \textbf{size} & \textbf{beds} \cdot \textbf{age} & \textbf{beds} \cdot \textbf{beds} \end{bmatrix}$$

Properties:

- Diagonal entries sum squares (always positive)
- Off-diagonal entries show feature correlations

Direct Solution Methods

Remember our plan:

1. Convert predictions into matrix operations ✓
2. Formulate optimization problem ✓
3. Convert to linear equations ✓
4. **Solve equations efficiently** ← We are here!

Today: Three methods for solving normal equations:

> Gaussian elimination, LU factorization, and QR factorization

Direct Solution Methods

Example with three features:

```
X = torch.tensor([
    [1500, 10, 3],    # house 1: size, age, bedrooms
    [2100, 2,  4],    # house 2
    [800,  50, 2],    # house 3
    [1800, 15, 3]     # house 4
])
y = torch.tensor([500000, 800000, 250000, 550000])
```

The normal equations $(X^T X)w = X^T y$ give us a system $Aw = b$ where:

- $A = X^T X$ is square matrix (3 × 3)
- $b = X^T y$ combines features and prices

Key considerations:

- Computational Efficiency

> (1) measured by number of arithmetic operations, (2) critical for large systems, (3) affects running time directly

- Numerical Stability

> (1) how measurement errors get amplified, (2) critical when features are correlated, (3) can make fast methods unreliable

Cost Analysis of Direct Methods

Two main costs:

1. Formation: Computing $X^T X$ and $X^T y$
2. Solution: Solving the resulting system

With n houses and p features:

- Computing $X^T X$: np^2 operations (p^2 dot products of size n vectors)
- Computing $X^T y$: np operations (p dot products of size n vectors)
- Solving $p \times p$ system: $\frac{2p^3}{3}$ operations (gaussian elimination)

Cost Analysis of Direct Methods

Which dominates depends on problem size:

```
# Case 1: Many houses, few features
n, p = 1000, 10  # 1000 houses, 10 features
formation_cost = n * p**2  # 100,000 operations
solution_cost = (2 * p**3) // 3  # ~667 operations
print("Case 1: Formation dominates")

# Case 2: Many houses, many features
```

Key insights:

- n >> p: Formation cost dominates
- n ≈ p: Both costs matter
- n << p: Solution cost dominates

Gaussian Elimination Steps

Step 1: First Elimination

Goal: Create zeros in first column below a_{11}

Compute multipliers:

$$m_{21} = \frac{a_{21}}{a_{11}} \quad \text{and} \quad m_{31} = \frac{a_{31}}{a_{11}}$$

After row operations:

$$\left[\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & b_1 \\ 0 & a'_{22} & a'_{23} & b'_2 \\ 0 & a'_{32} & a'_{33} & b'_3 \end{array}\right] \qquad (24 \text{ operations: } 12 \text{ multiplications, } 12 \text{ subtractions})$$

Gaussian Elimination: Step 2

Step 2: Second Elimination

Goal: Create zero in second column below a'_{22}

Compute multiplier:

$$m_{32} = \frac{a'_{32}}{a'_{22}}$$

After row operations:

$$\left[\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & b_1 \\ 0 & a'_{22} & a'_{23} & b'_2 \\ 0 & 0 & a''_{33} & b''_3 \end{array}\right] \quad (8 \text{ operations: } 4 \text{ multiplications, } 4 \text{ subtractions})$$

Gaussian Elimination: Back-substitution

Solve from bottom to top:

$$w_3 = \frac{b_3''}{a_{33}''} \qquad (1 \text{ division})$$

$$w_2 = \frac{b_2' - a_{23}' w_3}{a_{22}'} \qquad (2 \text{ ops} + 1 \text{ division})$$

$$w_1 = \frac{b_1 - a_{12} w_2 - a_{13} w_3}{a_{11}} \qquad (4 \text{ ops} + 1 \text{ division})$$

Total operations:

- 6 divisions
- 19 multiplications
- 19 additions/subtractions

LU Factorization

LU factorization:

$$A = LU = \begin{bmatrix} 1 & 0 & 0 \\ m_{21} & 1 & 0 \\ m_{31} & m_{32} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

Computational Cost Analysis

To find optimal weights:

1. Form normal equations: compute $X^T X$ and $X^T y$
2. Solve resulting system

With n houses and p features:

- Computing $X^T X$: np^2 operations
- Computing $X^T y$: np operations
- Solving $p \times p$ system: $\frac{2p^3}{3}$ operations

Example costs:

```
1000 houses, 10 features:
```

LU Factorization: Key Insight

Problem: Market changes mean new optimal weights

- 100 new houses sell
- Market conditions shift values
- Seasonal patterns affect prices

Can we avoid redoing all work?

LU factorization splits A into:

$$A = LU = \begin{bmatrix} 1 & 0 & 0 \\ m_{21} & 1 & 0 \\ m_{31} & m_{32} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

Solving with LU

Two steps:

1. Forward substitution ($Ly = b$):

$$
\begin{aligned}
y_1 &= b_1 \\
y_2 &= b_2 - m_{21} y_1 \\
y_3 &= b_3 - m_{31} y_1 - m_{32} y_2
\end{aligned}
$$

2. Back substitution ($Uw = y$):

$$
\begin{aligned}
w_3 &= y_3 / u_{33} \\
w_2 &= (y_2 - u_{23} w_3) / u_{22} \\
w_1 &= (y_1 - u_{12} w_2 - u_{13} w_3) / u_{11}
\end{aligned}
$$

LU Factorization: Cost Savings

When house prices change:

- Matrix $A = X^T X$ stays same (features unchanged)
- Only vector $b = X^T y$ changes (new prices)

Operation costs (1000 houses, 100 features):

```
Operation     | First Time  | Each Update
--------------|-------------|-------------
Factor A = LU| 667,000     | (done!)
Solve Ly = b | 5,000       | 5,000
Solve Uw = y | 5,000       | 5,000
Total         | 677,000     | 10,000
```

98.5% reduction in work for updates!

LU Factorization in Practice

```
def solve_with_lu(X, y):
    """Solve normal equations using LU factorization"""
    # Form normal equations
    XtX = X.T @ X
    Xty = X.T @ y

    # LU factorization
    L, U = torch.lu(XtX)

    # Solve Ly = Xty
    y = torch.triangular_solve(Xty, L, upper=False)[0]

    # Solve Uw = y
    w = torch.triangular_solve(y, U)[0]
    return w

# When only y changes, reuse L and U:
def update_solution(L, U, X, y_new):
    Xty = X.T @ y_new
    y = torch.triangular_solve(Xty, L, upper=False)[0]
    return torch.triangular_solve(y, U)[0]
```

Understanding Condition Number

The condition number κ measures sensitivity:

$$\kappa(A) = \|A\|\|A^{-1}\|$$

For normal equations:

$$\kappa(X^T X) = \kappa(X)^2$$

Error propagation:

$$\frac{\|\Delta w\|}{\|w\|} \leq \kappa(A) \frac{\|\Delta b\|}{\|b\|}$$

This is why QR is often better.

Stability in Practice

Example: House prices with correlated features

```
# Square footage and number of rooms are correlated
X = torch.tensor([
    [1500, 6],  # 1500 sq ft ≈ 6 rooms
    [2000, 8],  # 2000 sq ft ≈ 8 rooms
    [1800, 7]   # 1800 sq ft ≈ 7 rooms
])

# Small measurement error (0.1%)
X_noisy = X * (1 + 0.001 * torch.randn_like(X))

# Weights change dramatically
w1 = solve_normal_equations(X, y)      # [200, 1000]
w2 = solve_normal_equations(X_noisy, y)# [800, -500]

# But predictions stay similar
```

QR Factorization: A Better Way

Instead of forming $X^T X$, decompose X directly:

$$X = QR$$

where:

- Q: orthogonal matrix (perpendicular columns)
- R: upper triangular matrix

Clean implementation:

```
def solve_regression(X, y):
    """Solve linear regression using QR factorization"""
    Q, R = torch.qr(X)
```

Properties of QR

Q has special properties:

- Columns are perpendicular (orthogonal)
- Each column has length 1 (normalized)
- $Q^T Q = I$ (identity matrix)

Check orthogonality:

```
print("Q^T @ Q =\n", Q.T @ Q)

# Output:
# tensor([[1.0000, 0.0000, 0.0000],
#         [0.0000, 1.0000, 0.0000],
#         [0.0000, 0.0000, 1.0000]])
```

Solving with QR

Original problem: $Xw = y$

With QR: $(QR)w = y$

Multiply both sides by Q^T:

$$\begin{aligned} Q^T(QRw) &= Q^T y \\ (Q^T Q)Rw &= Q^T y \\ IRw &= Q^T y \\ Rw &= Q^T y \end{aligned}$$

Beautiful! We get triangular system without forming $X^T X$

Normal Equations: Squares condition number

QR vs Normal Equations

2.

Cost

- QR: ~$2np^2$ operations
- Normal Equations: ~np^2 operations

3.

Updates

- QR: Need full recomputation

The Limits of Direct Methods

Direct methods face hard constraint:

- Must complete entire computation before any solution
- Minutes of waiting for large problems
- Impractical for massive applications

This motivates iterative methods:

- Produce increasingly accurate predictions over time
- Trade perfect accuracy for faster results
- Essential for massive datasets

We'll explore these methods next lecture!