Data Distributions

What Problem This Solves

Distributions reveal patterns in how data behaves.

Understanding distributions helps you:

- Predict system behavior under load
- Set realistic SLAs and capacity plans
- Identify when something is abnormal
- Make better architectural decisions

The shape of your data tells you what to expect.

Intuition & Mental Model

Think: Histogram Shape Reveals System Behavior

```
Normal (bell curve):
  │    ╱‾╲     Most values near center
  │  ╱    ╲    Predictable, symmetric
  └─────────


Power law (long tail):
  │╲          Few huge values
  │ ╲___      Many small values
  └──────     Highly skewed

Bimodal (two peaks):
  │ ╱╲  ╱╲    Two distinct groups
  │╱  ╲╱  ╲   Mixed populations
  └──────────
```

Distribution = data's personality

Core Concepts

1. Normal Distribution (Gaussian)

The bell curve:

```
     │      ╱‾‾╲
     │    ╱      ╲       68% within ±1σ
     │   ╱   μ    ╲      95% within ±2σ
     │  ╱          ╲     99.7% within ±3σ
     └─────────────────
```

Properties:

- Symmetric around mean (μ)

- Mean = Median = Mode
- Defined by mean and standard deviation (σ)

Where it appears:

- Human heights, test scores
- Measurement errors
- Aggregates of many independent factors

In systems:

```javascript
// Response times sometimes normal-ish
// (when no outliers, consistent backend)
const responseTimes = [48, 50, 52, 51, 49, 53, 50, 52];
// Mean ≈ 50.6ms, StdDev ≈ 1.6ms

// Check if roughly normal
function checkNormality(data) {
  const avg = mean(data);
  const std = standardDeviation(data);

  const within1Std = data.filter(x =>
    Math.abs(x - avg) <= std
  ).length / data.length;

  const within2Std = data.filter(x =>
    Math.abs(x - avg) <= 2 * std
  ).length / data.length;

  return {
    within1Std: within1Std,      // Should be ~0.68
    within2Std: within2Std,      // Should be ~0.95
    looksNormal: within1Std > 0.6 && within1Std < 0.75
  };
}
```

Why it matters:

- Many statistical tests assume normality
- Lets you use mean and std dev confidently
- Can predict outlier frequency (3σ rule)

---

2. Power Law Distribution (Long Tail)

Few massive values, many tiny values:

```
|\
| \               "Rich get richer"
|  \___           "Winner takes most"
|      ----\___
└───────────────
```

Mathematical form:

```
P(x) ∝ x^(-α)

Where α typically 2-3
```

Classic examples:

- **Wealth distribution**: 1% owns 50% of wealth
- **City sizes**: Few megacities, many small towns
- **Word frequency**: "the" appears way more than "zephyr"
- **Web links**: Few sites get most links

In systems:

```
// API request distribution
const requestCounts = {
  '/health': 1000000,      // Health checks dominate
  '/api/users': 50000,
  '/api/posts': 20000,
  '/api/search': 10000,
  '/api/admin': 100,       // Long tail of rare endpoints
  // ... hundreds of endpoints with < 100 requests
};

// File sizes
const fileSizes = {
  'config.json': 2KB,      // Many small files
  'main.js': 50KB,
  'bundle.js': 500KB,
  'video.mp4': 50MB,       // Few huge files
  'database.db': 5GB
};
```

Pareto Principle (80/20 rule):

```
// 80% of traffic from 20% of endpoints
// 80% of errors from 20% of code
// 80% of revenue from 20% of customers

function findTop20Percent(items) {
  const sorted = items.sort((a, b) => b.value - a.value);
  const top20Index = Math.ceil(items.length * 0.2);
  const top20 = sorted.slice(0, top20Index);

  const top20Sum = sum(top20.map(x => x.value));
  const totalSum = sum(items.map(x => x.value));

  return {
    percentOfTotal: (top20Sum / totalSum) * 100,
    // Often ~80% for power law distributions
  };
}
```

Why it matters:

- Can't use mean (dominated by outliers)
- Caching helps (hot items get most traffic)
- Need to handle tail differently

---

3. Exponential Distribution

Time between events:

```
|\
| \              Memoryless
|  \___          "How long until next event?"
|      ---\___
└──────────────
```

Where it appears:

- Time until server failure
- Time between user arrivals
- Cache TTL effectiveness

```
// Exponential: P(X > t) = e^(-λt)
// λ = rate parameter (events per unit time)

function exponentialPDF(x, lambda) {
  return lambda * Math.exp(-lambda * x);
}

function exponentialCDF(x, lambda) {
  return 1 - Math.exp(-lambda * x);
}

// Example: Server failure rate
const failuresPerYear = 2; // λ = 2
const lambda = failuresPerYear / 365; // Per day

// Probability server survives next 30 days
const pSurvive30Days = Math.exp(-lambda * 30);
// ≈ 0.84 (84% chance)

// Expected time to failure
const expectedDays = 1 / lambda;
// = 182.5 days (6 months)
```

Memoryless property:

```
// Given server survived 100 days,
// probability it survives another 30 days
// is SAME as initial probability
// (It doesn't "wear out" in exponential model)
```

4. Poisson Distribution

Count of events in fixed time:

```
  |    /‾\        Discrete (counts)
  |  /    \       "How many events?"
  |/        \___
  └──────────────
    0 1 2 3 4 5
```

Where it appears:

- Requests per second
- Bugs per 1000 lines
- Errors per hour

```
// P(k events) = (λ^k * e^(-λ)) / k!
// λ = average rate

function poissonProbability(k, lambda) {
  return (lambda ** k * Math.exp(-lambda)) / factorial(k);
}

// Example: Average 100 requests/sec
const avgRate = 100;

// Probability of exactly 110 requests in next second
poissonProbability(110, avgRate); // ~0.024 (2.4%)

// Probability of > 120 (2+ std devs above mean)
let pOver120 = 0;
for (let k = 121; k < 200; k++) {
  pOver120 += poissonProbability(k, avgRate);
}
// Very small → would indicate anomaly
```

Capacity planning:

```
function requiredCapacity(avgRate, confidence = 0.99) {
  // Use normal approximation for large λ
  const mean = avgRate;
  const stdDev = Math.sqrt(avgRate);

  // Z-score for 99% confidence
  const z = 2.33;

  const capacity = mean + z * stdDev;
  return Math.ceil(capacity);
}
```

```
// Avg 100 req/sec, plan for 99th percentile
requiredCapacity(100, 0.99); // ~123 req/sec capacity needed
```

---

5. Bimodal Distribution

Two distinct peaks:

```
 │ ╱╲  ╱╲       Two populations
 │╱  ╲╱  ╲      Mixed groups
 └──────────
```

Where it appears:

- Mobile vs desktop user behavior
- Weekday vs weekend traffic
- Cached vs uncached response times

```
// Response times: cache hit vs miss
const responseTimes = [
  10, 12, 11, 13, 10,     // Cache hits (~12ms)
  150, 160, 155, 148, 162  // Cache misses (~155ms)
];

// Don't use single mean! (~86ms - meaningless)
// Instead: separate into two groups

function detectBimodal(data) {
  // Simple approach: k-means with k=2
  const sorted = [...data].sort((a, b) => a - b);
  const mid = Math.floor(sorted.length / 2);

  let group1 = sorted.slice(0, mid);
  let group2 = sorted.slice(mid);

  // Iterate to convergence (simplified)
  for (let i = 0; i < 10; i++) {
    const mean1 = mean(group1);
    const mean2 = mean(group2);
    const threshold = (mean1 + mean2) / 2;

    group1 = sorted.filter(x => x < threshold);
    group2 = sorted.filter(x => x >= threshold);
  }

  return {
    group1: { mean: mean(group1), count: group1.length },
    group2: { mean: mean(group2), count: group2.length }
  };
}
```

Why it matters:

- Single mean misleads
- Need to analyze groups separately
- Often indicates mixed populations

---

6. Uniform Distribution

All values equally likely:

```
│┌─────────┐     Flat
││         │     Evenly distributed
└┴─────────┴──
```

Where it appears:

- Random number generators
- Hash function outputs (ideal)
- Load balancing (goal)

```
// Perfect load balancer: uniform distribution
const serverLoads = [100, 98, 102, 99, 101]; // ← Good!
// vs
const badLoads = [50, 150, 120, 80, 200];   // ← Uneven

function checkUniformity(data) {
  const avg = mean(data);
  const maxDeviation = Math.max(...data.map(x => Math.abs(x - avg)));
  const coefficient = maxDeviation / avg;

  return {
    isUniform: coefficient < 0.1, // < 10% deviation
    coefficient: coefficient
  };
}
```

Software Engineering Connections

1. Latency Distribution Analysis

```
class LatencyMonitor {
  constructor() {
    this.samples = [];
  }

  record(latency) {
    this.samples.push(latency);
  }

  analyze() {
    const sorted = [...this.samples].sort((a, b) => a - b);
    return {
      mean: mean(sorted),
      median: median(sorted),
      p50: percentile(sorted, 50),
      p95: percentile(sorted, 95),
      p99: percentile(sorted, 99),
      p999: percentile(sorted, 99.9),

      // Shape analysis
      shape: this.detectShape(sorted)
    };
  }

  detectShape(sorted) {
    const avg = mean(sorted);
    const med = median(sorted);

    if (Math.abs(avg - med) / avg < 0.05) {
      return 'normal';
    } else if (avg > med * 1.2) {
      return 'right-skewed (long tail)';
    } else {
      return 'left-skewed';
    }
  }
}

// Use percentiles, not mean, for skewed distributions
```

---

2. Capacity Planning

```
function planCapacity(historicalData, targetPercentile = 95) {
  // Historical request rates (requests/sec)
  const rates = historicalData;

  // Assume Poisson distribution
  const avgRate = mean(rates);
  const stdDev = Math.sqrt(avgRate);

  // Z-score for target percentile
  const zScores = {
    90: 1.28,
    95: 1.645,
    99: 2.33,
    99.9: 3.09
  };

  const z = zScores[targetPercentile];
  const peakCapacity = avgRate + z * stdDev;
  return {
    average: avgRate,
    peak: Math.ceil(peakCapacity),
    recommendation: `Provision for ${Math.ceil(peakCapacity)} req/sec`,
    confidence: `${targetPercentile}% of time will be under capacity`
  };
}
```

---

3. Caching Strategy

```
// Power law: Cache hot items
function analyzeCacheStrategy(accessCounts) {
  const sorted = Object.entries(accessCounts)
    .sort(([,a], [,b]) => b - a);

  const totalAccesses = sum(sorted.map(([,count]) => count));

  // Find how many items for 80% of traffic
  let cumulativeAccesses = 0;
  let itemCount = 0;

  for (const [item, count] of sorted) {
    cumulativeAccesses += count;
    itemCount++;

    if (cumulativeAccesses >= totalAccesses * 0.8) {
      break;
    }
  }

  return {
    totalItems: sorted.length,
    itemsFor80Percent: itemCount,
    ratio: (itemCount / sorted.length * 100).toFixed(1) + '%',
    recommendation: `Cache top ${itemCount} items for 80% hit rate`
  };
}

// Typical result: 20% of items handle 80% of traffic (power law)
```

---

4. Anomaly Detection

```
class AnomalyDetector {
  constructor(historicalData) {
    this.mean = mean(historicalData);
    this.stdDev = standardDeviation(historicalData);
    this.distribution = this.detectDistribution(historicalData);
  }
  detectDistribution(data) {
    const avg = mean(data);
    const med = median(data);

    if (avg / med > 1.5) {
      return 'power-law';
    } else if (Math.abs(avg - med) / avg < 0.1) {
      return 'normal';
    } else {
      return 'skewed';
    }
  }

  isAnomaly(value) {
    if (this.distribution === 'normal') {
      // 3-sigma rule
      return Math.abs(value - this.mean) > 3 * this.stdDev;
    } else {
      // Use percentile for skewed
      return value > this.mean + 4 * this.stdDev;
    }
  }
}
```

---

5. Load Testing

```
// Generate realistic load (power law distribution)
function generateRealisticLoad(endpoints, totalRequests) {
  // Assign weights by power law
  const weights = endpoints.map((_, i) => 1 / ((i + 1) ** 1.5));
  const totalWeight = sum(weights);

  const normalizedWeights = weights.map(w => w / totalWeight);

  const distribution = endpoints.map((endpoint, i) => ({
    endpoint,
    requests: Math.floor(normalizedWeights[i] * totalRequests)
  }));

  return distribution;
}

// Results: First few endpoints get most traffic (realistic)
```

Common Misconceptions

❌ "All data is normally distributed"

Most real-world data is NOT normal:

- Response times: Right-skewed (long tail)
- User engagement: Power law
- Revenue: Power law (few big customers)

Always check distribution shape first.

❌ "Mean represents typical value"

For skewed distributions, median is better:

```
const salaries = [40K, 45K, 50K, 55K, 60K, 500K];
mean(salaries);   // $125K ← Misleading
median(salaries); // $52.5K ← Typical
```

❌ "Power law means unpredictable"

Opposite! Power laws are very predictable:

- 80/20 rule applies consistently
- Can plan caching, capacity around top items

❌ "Standard deviation only for normal distributions"

Std dev useful for any distribution, but interpretation differs:

- Normal: 68% within 1σ
- Others: Less predictable, use percentiles instead

❌ "Outliers are errors"

In power laws, "outliers" are normal:

- YouTube: Most videos get few views, few get millions
- Not errors—that's the distribution!

Practical Mini-Exercises

Exercise 1: Identify Distribution

```
const responseTime = [10, 12, 11, 13, 10, 11, 12, 500, 11, 10];
```

What distribution shape? Should you use mean or median?

▶ Solution

Exercise 2: Cache Sizing

```
const itemAccesses = {
  'item1': 10000,
  'item2': 5000,
  'item3': 2500,
  'item4': 1250,
  'item5': 625,
  // ... 95 more items with decreasing access counts
};
```

How many items to cache for 80% hit rate?

▶ Solution

Exercise 3: Detect Bimodal

```
const responseTimes = [
  12, 10, 11, 13, 150, 155, 148, 14, 12, 160,
  11, 145, 13, 10, 158, 12, 152, 11
];
```

Is this bimodal? What are the two groups?

▶ Solution

Summary Cheat Sheet

Distribution Types

Distribution	Shape	When It Appears	Use
Normal	Bell curve	Heights, errors, aggregates	Mean, std dev
Power Law	Long tail	Traffic, wealth, file sizes	Median, percentiles
Exponential	Decay curve	Time between events	Rate parameter
Poisson	Discrete peak	Event counts	Capacity planning
Bimodal	Two peaks	Mixed populations	Separate analysis
Uniform	Flat	Random, ideal load balance	Range

Quick Checks

```
// Check shape
const skew = (mean(data) - median(data)) / standardDeviation(data);
if (Math.abs(skew) < 0.5) return 'symmetric';
if (skew > 0.5) return 'right-skewed (long tail)';
return 'left-skewed';

// Check for outliers (IQR method)
const q1 = percentile(data, 25);
const q3 = percentile(data, 75);
const iqr = q3 - q1;
const outliers = data.filter(x =>
  x < q1 - 1.5*iqr || x > q3 + 1.5*iqr
);
```

```
// Power law check (80/20)
// If top 20% accounts for >70% of total → power law
```

Reporting Guidelines

```
// Normal distribution
report({ mean, stdDev });

// Skewed distribution
report({ median, p95, p99 });

// Bimodal
report({ group1Mean, group2Mean, splitRatio });

// Power law
report({ median, p50, p90, p99, top10PercentShare });
```

Next Steps

Understanding distributions helps you recognize patterns and make predictions about system behavior. You now know how to identify distribution types and choose appropriate statistical measures.

Next, we'll explore **financial mathematics**—understanding compound growth, time value of money, and making financial decisions.

Continue to: 10-financial-math.md