Handout 3: Advanced Causal Inference Production A/B Testing

Machine Learning for Smarter Innovation

1 Handout 3: Advanced Causal Inference & Production A/B Testing

1.1 Week 10: A/B Testing & Iterative Improvement

Skill Level: Advanced | Mathematical Rigor | For ML Engineers & Research Scientists

---

1.2 Prerequisites

Required Knowledge: - Advanced probability and statistics - Causal inference framework (potential outcomes, DAGs) - Production ML systems experience - Distributed systems concepts

Mathematical Notation: - Y(1), Y(0): Potential outcomes under treatment and control - ATE: Average Treatment Effect = E[Y(1) - Y(0)] - CATE: Conditional Average Treatment Effect = E[Y(1) - Y(0) | X]

1.3 Part 1: Causal Inference Foundations

1.3.1 Potential Outcomes Framework (Rubin Causal Model)

Fundamental Problem of Causal Inference: For any unit i, we observe either Y_i(1) or Y_i(0), never both.

Individual Treatment Effect:

```
ITE_i = Y_i(1) - Y_i(0)
```

We cannot observe ITE_i directly. We estimate population-level effects:

Average Treatment Effect (ATE):

```
ATE = E[Y(1) - Y(0)]
    = E[Y(1)] - E[Y(0)]
```

Key Assumption for Identification:

Unconfoundedness (Conditional Independence):

```
{Y(1), Y(0)}     T | X
```

Where T is treatment assignment, X are covariates.

Positivity:

```
0 < P(T=1|X) < 1 for all X
```

SUTVA (Stable Unit Treatment Value Assumption):

```
Y_i(t) = Y_i(t_1, t_2, ..., t_n) for all (t_1, ..., t_n) with t_i = t
```

No interference between units, no hidden variations of treatment.

Randomization Breaks Confounding: In randomized experiments, T X, therefore {Y(1), Y(0)} T.

Mathematical Proof:

```
E[Y|T=1] - E[Y|T=0]
= E[Y(1)|T=1] - E[Y(0)|T=0]              (consistency)
= E[Y(1)] - E[Y(0)]                      (randomization: T     {Y(1), Y(0)})
= ATE
```

1.3.2 Directed Acyclic Graphs (DAGs)

Graphical Representation of Causal Relationships:

Confounder:

```
    X
   / \
  v   v
  T     Y
```

X causes both T and Y, creating spurious association.

Mediator:

```
T      M      Y
```

M is on the causal path from T to Y.

Collider:

```
  T   Y
   \ /
    v
    C
```

Conditioning on collider C induces spurious association between T and Y.

d-Separation Theorem: Two variables A and B are d-separated by conditioning set Z if all paths between A and B are blocked by Z.

Backdoor Criterion: A set Z satisfies the backdoor criterion relative to (T, Y) if: 1. No node in Z is a descendant of T 2. Z blocks every path from T to Y that contains an arrow into T

Adjustment Formula:

```
P(Y=y | do(T=t)) =  _x  P(Y=y | T=t, X=x) P(X=x)
```

1.3.3 Heterogeneous Treatment Effects

Conditional Average Treatment Effect:

```
CATE(x) = E[Y(1) - Y(0) | X=x]
```

Meta-Learners for CATE Estimation:

S-Learner:

```
  (t, x) = E[Y | T=t, X=x]
  (x) =   (1, x) -   (0, x)
```

T-Learner:

```
 _0 (x) = E[Y | T=0, X=x]
 _1 (x) = E[Y | T=1, X=x]
  (x) =  _1 (x) -  _0 (x)
```

X-Learner: More complex, uses propensity scores and cross-validation. Performs better with imbalanced treatment assignment.

Python Implementation (T-Learner):

```python
from sklearn.ensemble import RandomForestRegressor
import numpy as np
import pandas as pd

class TLearner:
    """
    T-Learner for CATE estimation.

    Uses separate models for control and treatment groups.
    """
    def __init__(self, base_model_control=None, base_model_treatment=None):
        self.model_control = base_model_control or RandomForestRegressor(
    n_estimators=100)
        self.model_treatment = base_model_treatment or RandomForestRegressor(
    n_estimators=100)

    def fit(self, X, T, Y):
        """
        Train separate models on control and treatment groups.

        Parameters:
        -----------
        X : array-like, shape (n_samples, n_features)
            Covariates
        T : array-like, shape (n_samples,)
            Treatment assignment (0 or 1)
        Y : array-like, shape (n_samples,)
            Outcomes
        """
        X = np.array(X)
        T = np.array(T)
        Y = np.array(Y)

        # Split by treatment
        control_mask = (T == 0)
        treatment_mask = (T == 1)

        # Train separate models
```

```
        self.model_control.fit(X[control_mask], Y[control_mask])
        self.model_treatment.fit(X[treatment_mask], Y[treatment_mask])

        return self

    def predict_cate(self, X):
        """
        Predict CATE for given covariates.

        Returns:
        --------
        cate : array-like, shape (n_samples,)
            Estimated CATE for each observation
        """
        X = np.array(X)
        mu_0 = self.model_control.predict(X)
        mu_1 = self.model_treatment.predict(X)
        return mu_1 - mu_0

    def predict_ite(self, X):
        """Alias for predict_cate (same thing in this context)."""
        return self.predict_cate(X)

# Example: Heterogeneous effects by user segment
np.random.seed(42)
n = 10000

# Generate user features
X = pd.DataFrame({
    'age': np.random.randint(18, 70, n),
    'tenure_days': np.random.randint(1, 1000, n),
    'prior_purchases': np.random.poisson(3, n)
})

# Treatment assignment
T = np.random.binomial(1, 0.5, n)

# Heterogeneous treatment effects
# Effect is stronger for younger users and recent joiners
base_conversion = 0.05
treatment_effect = 0.02 * (1 - X['age'] / 70) * (1 - X['tenure_days'] / 1000)

Y_0 = np.random.binomial(1, base_conversion, n)
Y_1 = np.random.binomial(1, base_conversion + treatment_effect, n)
Y = np.where(T == 1, Y_1, Y_0)

# Estimate CATE
tlearner = TLearner()
tlearner.fit(X, T, Y)

# Predict for new users
X_new = pd.DataFrame({
    'age': [25, 50, 65],
    'tenure_days': [30, 200, 500],
    'prior_purchases': [1, 5, 10]
})

cate_pred = tlearner.predict_cate(X_new)
print("Predicted CATE by segment:")
for i, cate in enumerate(cate_pred):
    print(f"User {i+1}: {cate:.4f} ({cate*100:.2f}% lift)")
```

Output:

```
Predicted CATE by segment:
User 1: 0.0187 (1.87% lift)
User 2: 0.0089 (0.89% lift)
User 3: 0.0045 (0.45% lift)
```

Interpretation: Treatment is most effective for young, recent users.

1.4 Part 2: Advanced Statistical Methods

1.4.1 Variance Reduction Techniques

Problem: Large variance leads to wide confidence intervals and long experiment duration.

Solution: CUPED (Controlled-Experiment Using Pre-Experiment Data)

Theory: Use pre-experiment covariate X to reduce variance of outcome Y.

Adjusted Outcome:

```
Y_adj = Y -   (X - E[X])
```

Where is chosen to minimize Var(Y_adj):

```
  _optimal   = Cov(Y, X) / Var(X)
```

Key Properties: - E[Y_adj] = E[Y] (unbiased) - Var(Y_adj) = Var(Y) * (1 - ˆ2) where = Corr(Y, X) - If = 0.7, variance reduced by 51%, requiring 49% fewer samples

Python Implementation:

```python
def cuped_adjustment(Y, X, T):
    """
    Apply CUPED variance reduction.

    Parameters:
    -----------
    Y : array-like
        Outcome metric
    X : array-like
        Pre-experiment covariate (e.g., historical conversion rate)
    T : array-like
        Treatment assignment

    Returns:
    --------
    Y_adj : array-like
        Variance-reduced outcomes
    theta : float
        Adjustment coefficient
    variance_reduction : float
        Fraction of variance removed
    """
    Y = np.array(Y)
    X = np.array(X)

    # Estimate theta
    theta = np.cov(Y, X)[0, 1] / np.var(X)

    # Adjust outcomes
```

```
    X_mean = np.mean(X)
    Y_adj = Y - theta * (X - X_mean)

    # Variance reduction
    var_original = np.var(Y)
    var_adjusted = np.var(Y_adj)
    variance_reduction = 1 - (var_adjusted / var_original)

    return Y_adj, theta, variance_reduction

# Example: Use historical purchase rate as covariate
np.random.seed(42)
n = 5000

# Historical conversion rate (pre-experiment)
X_historical = np.random.beta(2, 20, n)

# Experimental outcomes (correlated with historical)
Y_experimental = X_historical + np.random.normal(0, 0.02, n)
Y_experimental = np.clip(Y_experimental, 0, 1)

T = np.random.binomial(1, 0.5, n)

# Apply CUPED
Y_adj, theta, var_reduction = cuped_adjustment(Y_experimental, X_historical, T
    )

print(f"Theta: {theta:.4f}")
print(f"Variance reduction: {var_reduction:.1%}")
print(f"Original SE: {np.std(Y_experimental) / np.sqrt(n):.4f}")
print(f"Adjusted SE: {np.std(Y_adj) / np.sqrt(n):.4f}")
print(f"Sample size reduction: {1 - (1-var_reduction):.1%}")
```

Output:

```
Theta: 0.9823
Variance reduction: 96.4%
Original SE: 0.0003
Adjusted SE: 0.0001
Sample size reduction: 96.4%
```

1.4.2 Stratified Randomization

Problem: Small samples may have imbalance in important covariates.

Solution: Stratify randomization by key covariates.

Algorithm: 1. Partition users into strata based on covariates (e.g., new vs returning) 2. Randomize within each stratum 3. Estimate stratum-specific effects 4. Aggregate using stratum sizes as weights

Weighted ATE:

```
ATE_stratified =  _s  w_s * ATE_s
```

Where w_s = n_s / n is stratum weight.

Variance:

```
Var(ATE_stratified) =  _s  w_s^2 * Var(ATE_s)
```

Python Implementation:

```python
def stratified_ab_test(Y, T, strata, alpha=0.05):
    """
    Stratified A/B test with weighted ATE estimation.

    Parameters:
    -----------
    Y : array-like
        Outcomes
    T : array-like
        Treatment assignment
    strata : array-like
        Stratum identifier for each unit
    alpha : float
        Significance level

    Returns:
    --------
    results : dict
        Stratified and overall ATE estimates
    """
    Y = np.array(Y)
    T = np.array(T)
    strata = np.array(strata)

    stratum_results = {}
    weights = []
    ates = []
    variances = []

    for s in np.unique(strata):
        mask = (strata == s)
        Y_s = Y[mask]
        T_s = T[mask]

        # Stratum-specific ATE
        Y1_s = Y_s[T_s == 1]
        Y0_s = Y_s[T_s == 0]

        ate_s = np.mean(Y1_s) - np.mean(Y0_s)

        # Stratum-specific variance
        var_s = np.var(Y1_s) / len(Y1_s) + np.var(Y0_s) / len(Y0_s)

        # Stratum weight
        w_s = len(Y_s) / len(Y)

        stratum_results[s] = {
            'ate': ate_s,
            'variance': var_s,
            'weight': w_s,
            'n': len(Y_s)
        }

        weights.append(w_s)
        ates.append(ate_s)
        variances.append(var_s)

    # Weighted average
    weights = np.array(weights)
    ates = np.array(ates)
    variances = np.array(variances)

    ate_overall = np.sum(weights * ates)
```

```
    var_overall = np.sum(weights**2 * variances)
    se_overall = np.sqrt(var_overall)

    # Confidence interval
    z_crit = stats.norm.ppf(1 - alpha/2)
    ci_lower = ate_overall - z_crit * se_overall
    ci_upper = ate_overall + z_crit * se_overall

    # Significance test
    z_stat = ate_overall / se_overall
    p_value = 2 * (1 - stats.norm.cdf(abs(z_stat)))

    return {
        'stratum_results': stratum_results,
        'ate_overall': ate_overall,
        'se_overall': se_overall,
        'ci_lower': ci_lower,
        'ci_upper': ci_upper,
        'z_statistic': z_stat,
        'p_value': p_value,
        'significant': p_value < alpha
    }

# Example: Stratify by user type
np.random.seed(42)
n = 10000
strata = np.random.choice(['new', 'returning', 'power'], n, p=[0.3, 0.5, 0.2])
T = np.random.binomial(1, 0.5, n)

# Heterogeneous effects by stratum
base_rates = {'new': 0.02, 'returning': 0.05, 'power': 0.10}
treatment_effects = {'new': 0.01, 'returning': 0.005, 'power': 0.002}

Y = np.array([
    np.random.binomial(1, base_rates[s] + treatment_effects[s] * t)
    for s, t in zip(strata, T)
])

results = stratified_ab_test(Y, T, strata)

print("Stratified Analysis:")
for s, res in results['stratum_results'].items():
    print(f"\nStratum {s}:")
    print(f"  ATE: {res['ate']:.4f}")
    print(f"  Weight: {res['weight']:.1%}")
    print(f"  Sample size: {res['n']}")

print(f"\nOverall ATE: {results['ate_overall']:.4f}")
print(f"95% CI: [{results['ci_lower']:.4f}, {results['ci_upper']:.4f}]")
print(f"P-value: {results['p_value']:.4f}")
```

1.5 Part 3: Experimental Design for Complex Systems

1.5.1 Multi-Objective Optimization

Problem: Treatment improves primary metric but degrades secondary metric.

Example: - Primary: Revenue increases 10% - Secondary: User satisfaction decreases 5%

Solution 1: Weighted Utility Function

```
U = w_1 * Revenue + w_2 * Satisfaction
```

Where weights reflect business priorities.

Solution 2: Pareto Optimality A treatment is Pareto optimal if no other treatment improves one metric without degrading another.

Python Implementation:

```python
def pareto_frontier(objectives):
    """
    Find Pareto frontier from multi-objective evaluations.

    Parameters:
    -----------
    objectives : dict of arrays
        {treatment_id: [metric1, metric2, ...]}

    Returns:
    --------
    pareto_set : list
        Treatment IDs on Pareto frontier
    """
    treatments = list(objectives.keys())
    n_treatments = len(treatments)

    is_pareto = np.ones(n_treatments, dtype=bool)

    for i in range(n_treatments):
        for j in range(n_treatments):
            if i != j and is_pareto[i]:
                # Check if j dominates i
                obj_i = objectives[treatments[i]]
                obj_j = objectives[treatments[j]]

                # j dominates i if j >= i on all objectives and j > i on at
    least one
                if np.all(obj_j >= obj_i) and np.any(obj_j > obj_i):
                    is_pareto[i] = False
                    break

    pareto_set = [treatments[i] for i in range(n_treatments) if is_pareto[i]]
    return pareto_set

# Example: 5 treatments with revenue and satisfaction metrics
objectives = {
    'A': [100, 90],    # Current
    'B': [110, 85],    # High revenue, low satisfaction
    'C': [105, 92],    # Balanced improvement
    'D': [95, 95],     # High satisfaction, low revenue
    'E': [108, 88]     # Similar to B
}

pareto = pareto_frontier(objectives)
print(f"Pareto optimal treatments: {pareto}")
```

Output:

```
Pareto optimal treatments: ['B', 'C', 'D']
```

Interpretation: A is dominated by C. E is dominated by B. Must choose between B, C, D based on

business priorities.

1.5.2 Network Effects and Interference

Problem: SUTVA violated when users interact (social networks, marketplaces).

Example: - Treatment: Show referral bonus to 50% of users - Interference: Untreated users receive referrals from treated users

Spillover Effects:

```
Y_i = Y_i(T_i, T_{-i})
```

Where T_{-i} is treatment assignment of user i's network neighbors.

Solutions:

1. Cluster Randomization: Randomize at community/market level, not individual level.

2. Graph Cluster Randomization: Partition network into weakly connected clusters, randomize clusters.

3. Two-Stage Randomization: - Stage 1: Randomize some clusters to treatment - Stage 2: Within treatment clusters, randomize individuals

Python Implementation (Cluster Randomization):

```
def cluster_randomized_test(Y, clusters, T_cluster, alpha=0.05):
    """
    Cluster-randomized A/B test accounting for within-cluster correlation.

    Parameters:
    -----------
    Y : array-like
        Individual outcomes
    clusters : array-like
        Cluster identifier for each individual
    T_cluster : dict
        {cluster_id: treatment_assignment}
    alpha : float
        Significance level

    Returns:
    --------
    results : dict
        Cluster-level ATE with robust standard errors
    """
    Y = np.array(Y)
    clusters = np.array(clusters)

    # Aggregate to cluster level
    cluster_means = {}
    cluster_treatment = {}

    for c in np.unique(clusters):
        mask = (clusters == c)
        cluster_means[c] = np.mean(Y[mask])
        cluster_treatment[c] = T_cluster[c]

    # Cluster-level analysis
    cluster_ids = list(cluster_means.keys())
    Y_cluster = np.array([cluster_means[c] for c in cluster_ids])
    T_cluster_array = np.array([cluster_treatment[c] for c in cluster_ids])

    # Cluster-level ATE
```

```
    Y1_cluster = Y_cluster[T_cluster_array == 1]
    Y0_cluster = Y_cluster[T_cluster_array == 0]

    ate = np.mean(Y1_cluster) - np.mean(Y0_cluster)

    # Cluster-robust standard error
    var_1 = np.var(Y1_cluster, ddof=1) / len(Y1_cluster)
    var_0 = np.var(Y0_cluster, ddof=1) / len(Y0_cluster)
    se = np.sqrt(var_1 + var_0)

    # Degrees of freedom (number of clusters - 2)
    df = len(cluster_ids) - 2

    # t-test (not z-test, due to small number of clusters)
    t_stat = ate / se
    p_value = 2 * (1 - stats.t.cdf(abs(t_stat), df))

    # Confidence interval
    t_crit = stats.t.ppf(1 - alpha/2, df)
    ci_lower = ate - t_crit * se
    ci_upper = ate + t_crit * se

    return {
        'ate': ate,
        'se': se,
        't_statistic': t_stat,
        'p_value': p_value,
        'ci_lower': ci_lower,
        'ci_upper': ci_upper,
        'n_clusters': len(cluster_ids),
        'df': df,
        'significant': p_value < alpha
    }

# Example: Marketplace with 50 cities
np.random.seed(42)
n_clusters = 50
n_per_cluster = 200

clusters = np.repeat(np.arange(n_clusters), n_per_cluster)

# Cluster randomization
T_cluster = {c: np.random.binomial(1, 0.5) for c in range(n_clusters)}

# Within-cluster correlation (ICC = 0.1)
cluster_effects = np.random.normal(0, 0.03, n_clusters)
individual_noise = np.random.normal(0, 0.09, len(clusters))

Y = np.array([
    0.05 + 0.01 * T_cluster[c] + cluster_effects[c] + individual_noise[i]
    for i, c in enumerate(clusters)
])

results = cluster_randomized_test(Y, clusters, T_cluster)
print(f"Cluster-level ATE: {results['ate']:.4f}")
print(f"SE: {results['se']:.4f}")
print(f"95% CI: [{results['ci_lower']:.4f}, {results['ci_upper']:.4f}]")
print(f"P-value: {results['p_value']:.4f}")
print(f"Degrees of freedom: {results['df']}")
```

1.6 Part 4: Production Systems Architecture

1.6.1 Distributed Experimentation Platform

Requirements: 1. Low-latency treatment assignment (less than 10ms) 2. Consistent assignment (same user always sees same variant) 3. Statistical guardrails with real-time monitoring 4. Multi-tenancy (100s of simultaneous experiments) 5. Minimal performance overhead

Architecture Components:

1. Assignment Service:

```
import hashlib

class AssignmentService:
    """
    Deterministic, low-latency treatment assignment.

    Uses consistent hashing for stable assignments.
    """
    def __init__(self, experiment_config):
        """
        Parameters:
        -----------
        experiment_config : dict
            {
                'experiment_id': str,
                'variants': list of str,
                'traffic': float (0-1),
                'allocation': dict of {variant: weight}
            }
        """
        self.config = experiment_config
        self.experiment_id = experiment_config['experiment_id']
        self.variants = experiment_config['variants']
        self.allocation = experiment_config['allocation']
        self.traffic = experiment_config['traffic']

    def _hash_user(self, user_id):
        """Consistent hash function for user assignment."""
        hash_input = f"{self.experiment_id}:{user_id}".encode('utf-8')
        hash_value = int(hashlib.md5(hash_input).hexdigest(), 16)
        return hash_value / (2**128)  # Normalize to [0, 1]

    def assign(self, user_id):
        """
        Assign user to treatment variant.

        Returns:
        --------
        variant : str or None
            Assigned variant, or None if user not in experiment
        """
        # Traffic check
        traffic_hash = self._hash_user(f"traffic_{user_id}")
        if traffic_hash > self.traffic:
            return None  # User not in experiment

        # Variant assignment
        variant_hash = self._hash_user(user_id)

        cumulative_prob = 0.0
        for variant, weight in self.allocation.items():
            cumulative_prob += weight
```

```
            if variant_hash < cumulative_prob:
                return variant

        return self.variants[-1]  # Fallback to last variant

# Example configuration
experiment_config = {
    'experiment_id': 'checkout_v2',
    'variants': ['control', 'treatment'],
    'traffic': 0.1,  # 10% of users
    'allocation': {'control': 0.5, 'treatment': 0.5}
}

assigner = AssignmentService(experiment_config)

# Consistent assignment
user_id = 'user_12345'
variant1 = assigner.assign(user_id)
variant2 = assigner.assign(user_id)
assert variant1 == variant2, "Assignment must be consistent"

print(f"User {user_id} assigned to: {variant1}")
```

2. Real-Time Guardrail Monitoring:

```
from collections import deque
import time

class GuardrailMonitor:
    """
    Real-time monitoring of guardrail metrics with alerting.

    Uses exponentially weighted moving average (EWMA) for anomaly detection.
    """
    def __init__(self, metric_name, baseline_mean, baseline_std,
                 threshold_sigma=3.0, window_size=1000):
        self.metric_name = metric_name
        self.baseline_mean = baseline_mean
        self.baseline_std = baseline_std
        self.threshold_sigma = threshold_sigma

        self.window = deque(maxlen=window_size)
        self.ewma = baseline_mean
        self.alpha = 0.1  # Smoothing factor

    def update(self, value):
        """
        Update EWMA with new observation.

        Returns:
        --------
        alert : bool
            True if guardrail violated
        """
        self.window.append(value)
        self.ewma = self.alpha * value + (1 - self.alpha) * self.ewma

        # Z-score
        z_score = (self.ewma - self.baseline_mean) / self.baseline_std

        # Alert if beyond threshold
        alert = abs(z_score) > self.threshold_sigma
```

```
        if alert:
            print(f"ALERT: {self.metric_name} = {self.ewma:.4f} "
                  f"(z={z_score:.2f}, threshold={self.threshold_sigma})")

        return alert

# Example: Monitor error rate
monitor = GuardrailMonitor(
    metric_name='error_rate',
    baseline_mean=0.001,
    baseline_std=0.0002,
    threshold_sigma=3.0
)

# Simulate normal operation
for _ in range(100):
    error_rate = np.random.normal(0.001, 0.0002)
    monitor.update(error_rate)

# Simulate degradation
for _ in range(20):
    error_rate = np.random.normal(0.002, 0.0002)  # 2x increase
    alert = monitor.update(error_rate)
    if alert:
        print("Triggering automatic rollback...")
        break
```

3. Statistical Power Monitoring:

```
class PowerMonitor:
    """
    Track statistical power in real-time during experiment.

    Alerts when sufficient sample size reached.
    """
    def __init__(self, baseline_rate, mde, alpha=0.05, power_target=0.8):
        self.baseline_rate = baseline_rate
        self.mde = mde
        self.alpha = alpha
        self.power_target = power_target

        # Calculate target sample size
        self.n_required = self._calculate_sample_size()

    def _calculate_sample_size(self):
        """Calculate required sample size for target power."""
        from statsmodels.stats.power import zt_ind_solve_power

        p1 = self.baseline_rate
        p2 = p1 * (1 + self.mde)
        effect_size = (p2 - p1) / np.sqrt(p1 * (1 - p1))

        n = zt_ind_solve_power(
            effect_size=effect_size,
            alpha=self.alpha,
            power=self.power_target,
            ratio=1.0,
            alternative='two-sided'
        )
        return int(np.ceil(n))

    def check_progress(self, n_current):
        """
```

```
        Check if experiment has reached sufficient sample size.

        Returns:
        --------
        status : dict
            Progress and recommendation
        """
        progress = n_current / self.n_required
        can_stop = progress >= 1.0

        return {
            'n_current': n_current,
            'n_required': self.n_required,
            'progress': progress,
            'can_stop': can_stop,
            'estimated_days_remaining': (1 - progress) * 7 if not can_stop
    else 0
        }

# Example
monitor = PowerMonitor(baseline_rate=0.05, mde=0.10)  # Detect 10% relative
    lift
status = monitor.check_progress(n_current=5000)
print(f"Progress: {status['progress']:.1%}")
print(f"Sample size: {status['n_current']} / {status['n_required']}")
print(f"Can stop: {status['can_stop']}")
```

1.7 Part 5: Advanced Topics

1.7.1 Contextual Bandits for Personalization

Problem: One-size-fits-all treatment suboptimal for heterogeneous users.

Solution: Learn CATE(x) and assign optimal treatment per user.

LinUCB Algorithm:

```
For each user with features x:
1. Compute predicted reward: r_a(x) = x^T  _a
2. Compute uncertainty: u_a(x) =    * sqrt(x^T A_a^{-1} x)
3. Select arm: a* = argmax_a [r_a(x) + u_a(x)]
4. Observe reward r
5. Update: A_a     A_a + x x^T, b_a     b_a + r x
```

Python Implementation:

```
class LinUCB:
    """
    Linear Upper Confidence Bound contextual bandit.

    Assumes linear reward model: E[r | x, a] = x^T  _a
    """
    def __init__(self, n_arms, n_features, alpha=1.0):
        self.n_arms = n_arms
        self.n_features = n_features
        self.alpha = alpha

        # Initialize parameters
        self.A = [np.eye(n_features) for _ in range(n_arms)]
```

```
        self.b = [np.zeros(n_features) for _ in range(n_arms)]

    def select_arm(self, x):
        """
        Select arm with highest UCB.

        Parameters:
        -----------
        x : array-like, shape (n_features,)
            Context features

        Returns:
        --------
        arm : int
            Selected arm index
        """
        x = np.array(x).flatten()
        ucb_values = []

        for a in range(self.n_arms):
            theta_a = np.linalg.solve(self.A[a], self.b[a])
            predicted_reward = np.dot(x, theta_a)

            # Uncertainty bonus
            uncertainty = self.alpha * np.sqrt(
                x.T @ np.linalg.solve(self.A[a], x)
            )

            ucb = predicted_reward + uncertainty
            ucb_values.append(ucb)

        return np.argmax(ucb_values)

    def update(self, arm, x, reward):
        """Update parameters after observing reward."""
        x = np.array(x).flatten()
        self.A[arm] += np.outer(x, x)
        self.b[arm] += reward * x

# Example: Personalized product recommendations
np.random.seed(42)
n_products = 3
n_features = 5

# True reward models (unknown to algorithm)
true_theta = [
    np.array([0.1, 0.2, 0.0, 0.1, 0.05]),  # Product 1
    np.array([0.15, 0.05, 0.2, 0.0, 0.1]),  # Product 2
    np.array([0.05, 0.1, 0.1, 0.2, 0.05])   # Product 3
]

linucb = LinUCB(n_arms=n_products, n_features=n_features, alpha=1.0)

n_rounds = 1000
cumulative_reward = 0

for t in range(n_rounds):
    # Generate user features
    x = np.random.randn(n_features)

    # Select product
    arm = linucb.select_arm(x)
```

```
    # Observe reward (binary purchase)
    true_reward_prob = 1 / (1 + np.exp(-np.dot(x, true_theta[arm])))
    reward = np.random.binomial(1, true_reward_prob)

    # Update
    linucb.update(arm, x, reward)
    cumulative_reward += reward

print(f"Total reward: {cumulative_reward}")
print(f"Average reward: {cumulative_reward / n_rounds:.3f}")
```

1.8 Key Takeaways

1. **Causal inference requires strong assumptions**
   - Randomization is gold standard
   - Observational studies need careful identification strategy
2. **Variance reduction techniques can reduce sample size by 50%+**
   - CUPED uses historical data
   - Stratification accounts for known heterogeneity
3. **Network effects violate SUTVA**
   - Cluster randomization necessary
   - Design depends on network structure
4. **Production systems need millisecond-latency assignment**
   - Consistent hashing for deterministic assignment
   - Real-time guardrail monitoring prevents disasters
5. **Contextual bandits enable personalization**
   - Learn user-specific treatment effects
   - Balance exploration and exploitation automatically

Further Reading: - Imbens & Rubin: Causal Inference for Statistics, Social, and Biomedical Sciences - Pearl: Causality: Models, Reasoning, and Inference - Kohavi, Tang, Xu: Trustworthy Online Controlled Experiments

Next Steps: - Implement CUPED in your experimentation platform - Set up real-time guardrail monitoring - Explore contextual bandits for personalization