A/B Testing - Advanced Handout

Machine Learning for Smarter Innovation

1 A/B Testing - Advanced Handout

Target Audience: Data scientists and ML engineers **Duration**: 90 minutes reading **Level**: Advanced (mathematical foundations + production systems)

1.1 Mathematical Foundations

1.1.1 Potential Outcomes Framework

The Rubin Causal Model defines causation through potential outcomes. For unit i with treatment indicator $T_i \in \{0, 1\}$, define: - $Y_i(1)$: outcome if treated - $Y_i(0)$: outcome if control

The **Individual Treatment Effect** is:

$$\tau_i = Y_i(1) - Y_i(0)$$

The **fundamental problem of causal inference**: we observe only one potential outcome per unit. The observed outcome is:

$$Y_i = T_i \cdot Y_i(1) + (1 - T_i) \cdot Y_i(0)$$

1.1.2 Average Treatment Effect

The **Average Treatment Effect (ATE)** is:

$$\tau = \mathbb{E}[Y(1) - Y(0)] = \mathbb{E}[Y(1)] - \mathbb{E}[Y(0)]$$

The naive estimator $\mathbb{E}[Y|T = 1] - \mathbb{E}[Y|T = 0]$ equals ATE only under specific conditions.

Identification under Randomization:

$$\mathbb{E}[Y|T = 1] - \mathbb{E}[Y|T = 0] = \mathbb{E}[Y(1)|T = 1] - \mathbb{E}[Y(0)|T = 0]$$

With random assignment, $T \perp \{Y(0), Y(1)\}$, so:

$$= \mathbb{E}[Y(1)] - \mathbb{E}[Y(0)] = \tau$$

1.1.3 Key Assumptions

SUTVA (Stable Unit Treatment Value Assumption): 1. No interference: $Y_i(T_1, \ldots, T_n) = Y_i(T_i)$ 2. No hidden versions: treatment is well-defined

Positivity: $0 < P(T = 1|X) < 1$ for all X in the support
Unconfoundedness: $(Y(0), Y(1)) \perp T|X$

1.1.4 Conditional Average Treatment Effect

The **CATE** allows for heterogeneous effects:

$$\tau(x) = \mathbb{E}[Y(1) - Y(0)|X = x]$$

Under unconfoundedness, CATE is identified as:

$$\tau(x) = \mathbb{E}[Y|T = 1, X = x] - \mathbb{E}[Y|T = 0, X = x]$$

1.2 Statistical Inference

1.2.1 Two-Sample Test

For binary outcomes with sample proportions $\hat{p}_1$ and $\hat{p}_0$:

$$\hat{\tau} = \hat{p}_1 - \hat{p}_0$$

Under the null hypothesis $H_0 : p_1 = p_0 = p$:

$$\text{Var}(\hat{\tau}) = p(1 - p)\left(\frac{1}{n_1} + \frac{1}{n_0}\right)$$

The pooled estimator $\hat{p} = \frac{n_1\hat{p}_1 + n_0\hat{p}_0}{n_1 + n_0}$ gives the test statistic:

$$z = \frac{\hat{p}_1 - \hat{p}_0}{\sqrt{\hat{p}(1 - \hat{p})(1/n_1 + 1/n_0)}} \xrightarrow{d} N(0, 1)$$

1.2.2 Sample Size Calculation

For detecting effect δ with power $1 - \beta$ at significance α:

$$n = \frac{(z_{\alpha/2} + z_\beta)^2 \cdot 2\sigma^2}{\delta^2}$$

For proportions with p_0 in control and $p_1 = p_0(1 + \delta_{rel})$ in treatment:

$$n = \frac{(z_{\alpha/2}\sqrt{2\bar{p}(1 - \bar{p})} + z_\beta\sqrt{p_0(1 - p_0) + p_1(1 - p_1)})^2}{(p_1 - p_0)^2}$$

where $\bar{p} = (p_0 + p_1)/2$.

1.2.3 Confidence Intervals

The $(1 - \alpha)$ confidence interval for the treatment effect:

$$\hat{\tau} \pm z_{\alpha/2} \cdot \sqrt{\frac{\hat{p}_1(1 - \hat{p}_1)}{n_1} + \frac{\hat{p}_0(1 - \hat{p}_0)}{n_0}}$$

For continuous outcomes:

$$\hat{\tau} \pm t_{\alpha/2,df} \cdot \sqrt{\frac{s_1^2}{n_1} + \frac{s_0^2}{n_0}}$$

with Welch-Satterthwaite degrees of freedom.

1.3 Sequential Testing

1.3.1 The Peeking Problem

Repeated significance testing inflates Type I error. If we test k times at level α:

$$P(\text{at least one false positive}) = 1 - (1-\alpha)^k \approx k\alpha$$

1.3.2 Alpha Spending Functions

The O'Brien-Fleming boundary controls overall α across interim analyses. At analysis j of K total, the boundary is:

$$z_j^* = z_{\alpha/2} \cdot \sqrt{K/j}$$

Pocock boundaries use constant $z_j^* = z_{\alpha'/2}$ where α' is adjusted.

1.3.3 Always-Valid Inference

Sequential probability ratio tests provide anytime-valid p-values. For the mixture martingale:

$$M_n = \int \prod_{i=1}^{n} \frac{f(X_i;\theta)}{f(X_i;\theta_0)} \pi(\theta) d\theta$$

The process $(1/M_n)_n$ is a test supermartingale. Stopping when $1/M_n < \alpha$ controls Type I error at α.

Confidence Sequences: A $(1-\alpha)$ confidence sequence $\{C_n\}$ satisfies:

$$P(\theta \in C_n \text{ for all } n) \geq 1 - \alpha$$

1.4 Variance Reduction

1.4.1 CUPED (Controlled-Experiment Using Pre-Experiment Data)

Using pre-experiment covariate X to reduce variance:

$$Y_{adj} = Y - \theta(X - \mathbb{E}[X])$$

The optimal coefficient minimizing variance:

$$\theta^* = \frac{\text{Cov}(Y, X)}{\text{Var}(X)}$$

Variance reduction factor:

$$\frac{\text{Var}(Y_{adj})}{\text{Var}(Y)} = 1 - \rho_{Y,X}^2$$

where $\rho_{Y,X}$ is the correlation between Y and X.

If $\rho = 0.7$, variance reduces by 51%, equivalent to doubling sample size.

1.4.2 Stratified Estimation

With strata $s \in \{1, \ldots, S\}$ and stratum proportions $w_s = n_s/n$:

$$\hat{\tau}_{strat} = \sum_{s=1}^{S} w_s \hat{\tau}_s$$

The variance is:

$$\text{Var}(\hat{\tau}_{strat}) = \sum_{s=1}^{S} w_s^2 \text{Var}(\hat{\tau}_s)$$

Stratification never increases variance and reduces it when treatment effects vary across strata.

1.4.3 Regression Adjustment

Lin's estimator adds treatment-covariate interactions:

$$Y_i = \alpha + \tau T_i + \beta X_i + \gamma T_i (X_i - \bar{X}) + \epsilon_i$$

This is consistent for ATE regardless of model specification when treatment is randomized.

1.5 Implementation: Statistical Analysis Engine

```
"""
Production A/B testing statistical analysis with variance reduction.
"""

import numpy as np
from typing import Dict, Optional, Tuple
from dataclasses import dataclass
from scipy import stats
from scipy.special import expit


@dataclass
class ABTestResult:
    """Complete A/B test analysis result."""
    treatment_mean: float
    control_mean: float
    effect: float
    relative_effect: float
    standard_error: float
    confidence_interval: Tuple[float, float]
    z_statistic: float
    p_value: float
```

```
    significant: bool
    sample_sizes: Dict[str, int]
    power: float


class ABTestAnalyzer:
    """
    Comprehensive A/B test analysis with multiple methods.
    """

    def __init__(self, alpha: float = 0.05, two_sided: bool = True):
        self.alpha = alpha
        self.two_sided = two_sided

    def analyze(
        self,
        y_treatment: np.ndarray,
        y_control: np.ndarray,
        method: str = "standard"
    ) -> ABTestResult:
        """
        Analyze A/B test results.

        Args:
            y_treatment: Outcomes for treatment group
            y_control: Outcomes for control group
            method: "standard", "welch", or "bootstrap"
        """
        n_t = len(y_treatment)
        n_c = len(y_control)

        mean_t = np.mean(y_treatment)
        mean_c = np.mean(y_control)
        effect = mean_t - mean_c
        relative_effect = effect / mean_c if mean_c != 0 else np.inf

        if method == "standard":
            se, z_stat, p_val = self._standard_test(y_treatment, y_control)
        elif method == "welch":
            se, z_stat, p_val = self._welch_test(y_treatment, y_control)
        elif method == "bootstrap":
            se, z_stat, p_val = self._bootstrap_test(y_treatment, y_control)
        else:
            raise ValueError(f"Unknown method: {method}")

        # Confidence interval
        z_crit = stats.norm.ppf(1 - self.alpha/2) if self.two_sided else stats
    .norm.ppf(1 - self.alpha)
        ci = (effect - z_crit * se, effect + z_crit * se)

        # Power calculation (post-hoc)
        power = self._calculate_power(effect, se)

        return ABTestResult(
            treatment_mean=mean_t,
            control_mean=mean_c,
            effect=effect,
            relative_effect=relative_effect,
            standard_error=se,
            confidence_interval=ci,
            z_statistic=z_stat,
            p_value=p_val,
            significant=p_val < self.alpha,
```

```
            sample_sizes={"treatment": n_t, "control": n_c},
            power=power
        )

    def _standard_test(
        self,
        y_t: np.ndarray,
        y_c: np.ndarray
    ) -> Tuple[float, float, float]:
        """Standard two-sample z-test."""
        n_t, n_c = len(y_t), len(y_c)
        var_t = np.var(y_t, ddof=1)
        var_c = np.var(y_c, ddof=1)

        se = np.sqrt(var_t/n_t + var_c/n_c)
        effect = np.mean(y_t) - np.mean(y_c)
        z_stat = effect / se

        if self.two_sided:
            p_val = 2 * (1 - stats.norm.cdf(abs(z_stat)))
        else:
            p_val = 1 - stats.norm.cdf(z_stat)

        return se, z_stat, p_val

    def _welch_test(
        self,
        y_t: np.ndarray,
        y_c: np.ndarray
    ) -> Tuple[float, float, float]:
        """Welch's t-test for unequal variances."""
        stat, p_val = stats.ttest_ind(y_t, y_c, equal_var=False)

        n_t, n_c = len(y_t), len(y_c)
        var_t = np.var(y_t, ddof=1)
        var_c = np.var(y_c, ddof=1)
        se = np.sqrt(var_t/n_t + var_c/n_c)

        return se, stat, p_val

    def _bootstrap_test(
        self,
        y_t: np.ndarray,
        y_c: np.ndarray,
        n_bootstrap: int = 10000
    ) -> Tuple[float, float, float]:
        """Bootstrap confidence interval and p-value."""
        observed_effect = np.mean(y_t) - np.mean(y_c)

        # Bootstrap distribution of effect
        effects = []
        for _ in range(n_bootstrap):
            boot_t = np.random.choice(y_t, size=len(y_t), replace=True)
            boot_c = np.random.choice(y_c, size=len(y_c), replace=True)
            effects.append(np.mean(boot_t) - np.mean(boot_c))

        effects = np.array(effects)
        se = np.std(effects)

        # P-value via shift
        combined = np.concatenate([y_t, y_c])
        null_effects = []
        for _ in range(n_bootstrap):
```

```
            perm = np.random.permutation(combined)
            null_effects.append(
                np.mean(perm[:len(y_t)]) - np.mean(perm[len(y_t):])
            )

        p_val = np.mean(np.abs(null_effects) >= abs(observed_effect))

        z_stat = observed_effect / se if se > 0 else 0

        return se, z_stat, p_val

    def _calculate_power(self, effect: float, se: float) -> float:
        """Calculate achieved power given observed effect and SE."""
        if se == 0:
            return 1.0

        z_crit = stats.norm.ppf(1 - self.alpha/2)
        z_effect = abs(effect) / se

        power = 1 - stats.norm.cdf(z_crit - z_effect)
        return power


class CUPEDAdjuster:
    """
    CUPED variance reduction using pre-experiment covariates.
    """

    def __init__(self):
        self.theta = None
        self.x_mean = None

    def fit(self, y: np.ndarray, x: np.ndarray):
        """Estimate optimal adjustment coefficient."""
        self.theta = np.cov(y, x)[0, 1] / np.var(x)
        self.x_mean = np.mean(x)
        return self

    def transform(self, y: np.ndarray, x: np.ndarray) -> np.ndarray:
        """Apply CUPED adjustment."""
        return y - self.theta * (x - self.x_mean)

    def fit_transform(self, y: np.ndarray, x: np.ndarray) -> np.ndarray:
        """Fit and transform in one step."""
        self.fit(y, x)
        return self.transform(y, x)

    def variance_reduction(self, y: np.ndarray, x: np.ndarray) -> float:
        """Calculate variance reduction achieved."""
        y_adj = self.transform(y, x)
        return 1 - np.var(y_adj) / np.var(y)


class StratifiedAnalyzer:
    """
    Stratified A/B test analysis.
    """

    def __init__(self, alpha: float = 0.05):
        self.alpha = alpha

    def analyze(
        self,
```

```
        y: np.ndarray,
        t: np.ndarray,
        strata: np.ndarray
    ) -> Dict:
        """
        Stratified analysis with stratum-specific and overall effects.
        """
        unique_strata = np.unique(strata)
        n_total = len(y)

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

        for s in unique_strata:
            mask = (strata == s)
            y_s = y[mask]
            t_s = t[mask]

            # Stratum-specific analysis
            y_t = y_s[t_s == 1]
            y_c = y_s[t_s == 0]

            if len(y_t) == 0 or len(y_c) == 0:
                continue

            effect_s = np.mean(y_t) - np.mean(y_c)
            var_s = np.var(y_t, ddof=1)/len(y_t) + np.var(y_c, ddof=1)/len(y_c
    )
            weight_s = len(y_s) / n_total

            stratum_results[s] = {
                "effect": effect_s,
                "variance": var_s,
                "se": np.sqrt(var_s),
                "weight": weight_s,
                "n": len(y_s)
            }

            weights.append(weight_s)
            effects.append(effect_s)
            variances.append(var_s)

        # Overall weighted effect
        weights = np.array(weights)
        effects = np.array(effects)
        variances = np.array(variances)

        overall_effect = np.sum(weights * effects)
        overall_var = np.sum(weights**2 * variances)
        overall_se = np.sqrt(overall_var)

        # Confidence interval and test
        z_crit = stats.norm.ppf(1 - self.alpha/2)
        ci = (overall_effect - z_crit * overall_se,
              overall_effect + z_crit * overall_se)

        z_stat = overall_effect / overall_se if overall_se > 0 else 0
        p_value = 2 * (1 - stats.norm.cdf(abs(z_stat)))

        return {
            "stratum_results": stratum_results,
```

```
            "overall_effect": overall_effect,
            "overall_se": overall_se,
            "ci": ci,
            "z_statistic": z_stat,
            "p_value": p_value,
            "significant": p_value < self.alpha
        }
```

1.6 Implementation: Sequential Testing

```
"""
Sequential testing with valid stopping rules.
"""

import numpy as np
from typing import Optional, List, Tuple
from dataclasses import dataclass


@dataclass
class SequentialResult:
    """Result from sequential analysis."""
    stopped: bool
    decision: str  # "reject_null", "accept_null", "continue"
    p_value: float
    effect_estimate: float
    confidence_interval: Tuple[float, float]
    n_samples: int
    boundary_crossed: str  # "upper", "lower", or "none"


class SequentialTester:
    """
    Group sequential testing with spending functions.
    """

    def __init__(
        self,
        alpha: float = 0.05,
        n_analyses: int = 5,
        spending: str = "obrien_fleming"
    ):
        self.alpha = alpha
        self.n_analyses = n_analyses
        self.spending = spending

        # Compute spending function values
        self.alpha_spent = self._compute_spending()
        self.current_analysis = 0
        self.cumulative_alpha = 0.0

    def _compute_spending(self) -> List[float]:
        """Compute alpha to spend at each analysis."""
        if self.spending == "obrien_fleming":
            # O'Brien-Fleming: conservative early, liberal late
            info_fracs = np.linspace(1/self.n_analyses, 1.0, self.n_analyses)
            alpha_spent = [
                2 * (1 - stats.norm.cdf(
                    stats.norm.ppf(1 - self.alpha/2) / np.sqrt(t)
```

```
                ))
                for t in info_fracs
            ]
            # Convert to incremental spending
            incremental = [alpha_spent[0]]
            for i in range(1, len(alpha_spent)):
                incremental.append(alpha_spent[i] - alpha_spent[i-1])
            return incremental

        elif self.spending == "pocock":
            # Pocock: equal spending
            return [self.alpha / self.n_analyses] * self.n_analyses

        elif self.spending == "haybittle_peto":
            # Haybittle-Peto: 0.001 for interim, rest at final
            interim_alpha = 0.001
            return [interim_alpha] * (self.n_analyses - 1) + [
                self.alpha - (self.n_analyses - 1) * interim_alpha
            ]

        else:
            raise ValueError(f"Unknown spending function: {self.spending}")

    def analyze(
        self,
        y_treatment: np.ndarray,
        y_control: np.ndarray
    ) -> SequentialResult:
        """
        Perform interim analysis with sequential boundaries.
        """
        if self.current_analysis >= self.n_analyses:
            raise ValueError("All analyses exhausted")

        # Get current alpha threshold
        current_alpha = self.alpha_spent[self.current_analysis]
        self.cumulative_alpha += current_alpha

        # Compute test statistic
        n_t, n_c = len(y_treatment), len(y_control)
        mean_t, mean_c = np.mean(y_treatment), np.mean(y_control)
        effect = mean_t - mean_c

        var_t = np.var(y_treatment, ddof=1)
        var_c = np.var(y_control, ddof=1)
        se = np.sqrt(var_t/n_t + var_c/n_c)

        z_stat = effect / se if se > 0 else 0

        # Two-sided boundary
        z_boundary = stats.norm.ppf(1 - current_alpha/2)

        # Check boundaries
        if abs(z_stat) > z_boundary:
            stopped = True
            decision = "reject_null"
            boundary_crossed = "upper" if z_stat > 0 else "lower"
        elif self.current_analysis == self.n_analyses - 1:
            stopped = True
            decision = "accept_null"
            boundary_crossed = "none"
        else:
            stopped = False
```

```
            decision = "continue"
            boundary_crossed = "none"

        # P-value (adjusted for sequential nature)
        p_value = 2 * (1 - stats.norm.cdf(abs(z_stat)))

        # Confidence interval (using current boundary)
        ci = (effect - z_boundary * se, effect + z_boundary * se)

        self.current_analysis += 1

        return SequentialResult(
            stopped=stopped,
            decision=decision,
            p_value=p_value,
            effect_estimate=effect,
            confidence_interval=ci,
            n_samples=n_t + n_c,
            boundary_crossed=boundary_crossed
        )

    def reset(self):
        """Reset for new experiment."""
        self.current_analysis = 0
        self.cumulative_alpha = 0.0


class MixtureSPRT:
    """
    Mixture Sequential Probability Ratio Test.
    Provides always-valid inference.
    """

    def __init__(
        self,
        alpha: float = 0.05,
        prior_scale: float = 1.0
    ):
        self.alpha = alpha
        self.prior_scale = prior_scale
        self.log_likelihood_ratio = 0.0
        self.n_samples = 0

    def update(
        self,
        y_treatment: np.ndarray,
        y_control: np.ndarray
    ) -> SequentialResult:
        """
        Update with new data and check stopping rule.
        """
        # Sufficient statistics
        n_t = len(y_treatment)
        n_c = len(y_control)
        sum_t = np.sum(y_treatment)
        sum_c = np.sum(y_control)

        self.n_samples += n_t + n_c

        # For binary outcomes, use beta-binomial mixture
        # Log Bayes factor against null
        mean_t = sum_t / n_t if n_t > 0 else 0.5
        mean_c = sum_c / n_c if n_c > 0 else 0.5
```

```
        effect = mean_t - mean_c
        se = np.sqrt(mean_t*(1-mean_t)/n_t + mean_c*(1-mean_c)/n_c)

        # Simplified mixture likelihood ratio
        if se > 0:
            z = effect / se
            # Against point null
            log_bf = 0.5 * z**2 - 0.5 * np.log(1 + self.n_samples * self.
    prior_scale**2)
        else:
            log_bf = 0

        self.log_likelihood_ratio = log_bf

        # Stopping rule: reject when BF > 1/alpha
        threshold = -np.log(self.alpha)
        stopped = self.log_likelihood_ratio > threshold

        # Always-valid p-value
        p_value = min(1.0, np.exp(-self.log_likelihood_ratio))

        # Confidence sequence
        width = self.prior_scale * np.sqrt(
            2 * (np.log(1/self.alpha) + 0.5 * np.log(self.n_samples + 1)) /
            self.n_samples
        ) if self.n_samples > 0 else np.inf

        ci = (effect - width, effect + width)

        return SequentialResult(
            stopped=stopped,
            decision="reject_null" if stopped else "continue",
            p_value=p_value,
            effect_estimate=effect,
            confidence_interval=ci,
            n_samples=self.n_samples,
            boundary_crossed="upper" if stopped and effect > 0 else "lower" if
     stopped else "none"
        )

    def reset(self):
        """Reset for new experiment."""
        self.log_likelihood_ratio = 0.0
        self.n_samples = 0
```

1.7 Implementation: Heterogeneous Treatment Effects

```
"""
CATE estimation using meta-learners.
"""

import numpy as np
from sklearn.base import clone, BaseEstimator
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.model_selection import cross_val_predict
from typing import Optional
```

```
class TLearner:
    """
    T-Learner: Separate models for treatment and control.
    """

    def __init__(
        self,
        base_model: Optional[BaseEstimator] = None
    ):
        self.base_model = base_model or RandomForestRegressor(n_estimators
    =100)
        self.model_treatment = None
        self.model_control = None

    def fit(self, X: np.ndarray, T: np.ndarray, Y: np.ndarray):
        """Fit separate models on each treatment arm."""
        X = np.asarray(X)
        T = np.asarray(T)
        Y = np.asarray(Y)

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

        self.model_treatment = clone(self.base_model)
        self.model_control = clone(self.base_model)

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

        return self

    def predict(self, X: np.ndarray) -> np.ndarray:
        """Predict CATE for given covariates."""
        X = np.asarray(X)
        mu_1 = self.model_treatment.predict(X)
        mu_0 = self.model_control.predict(X)
        return mu_1 - mu_0


class SLearner:
    """
    S-Learner: Single model with treatment as feature.
    """

    def __init__(
        self,
        base_model: Optional[BaseEstimator] = None
    ):
        self.base_model = base_model or GradientBoostingRegressor(n_estimators
    =100)
        self.model = None

    def fit(self, X: np.ndarray, T: np.ndarray, Y: np.ndarray):
        """Fit single model including treatment indicator."""
        X = np.asarray(X)
        T = np.asarray(T).reshape(-1, 1)
        Y = np.asarray(Y)

        X_augmented = np.hstack([X, T])
        self.model = clone(self.base_model)
        self.model.fit(X_augmented, Y)
```

```
        return self

    def predict(self, X: np.ndarray) -> np.ndarray:
        """Predict CATE by contrasting predictions."""
        X = np.asarray(X)
        n = X.shape[0]

        X_treat = np.hstack([X, np.ones((n, 1))])
        X_control = np.hstack([X, np.zeros((n, 1))])

        mu_1 = self.model.predict(X_treat)
        mu_0 = self.model.predict(X_control)

        return mu_1 - mu_0


class XLearner:
    """
    X-Learner: Uses propensity scores and cross-fitting.
    Better for imbalanced treatment assignment.
    """

    def __init__(
        self,
        outcome_model: Optional[BaseEstimator] = None,
        effect_model: Optional[BaseEstimator] = None,
        propensity_model: Optional[BaseEstimator] = None
    ):
        self.outcome_model = outcome_model or RandomForestRegressor(
    n_estimators=100)
        self.effect_model = effect_model or RandomForestRegressor(n_estimators
    =100)
        self.propensity_model = propensity_model

        self.mu_0 = None
        self.mu_1 = None
        self.tau_0 = None
        self.tau_1 = None
        self.propensity = None

    def fit(self, X: np.ndarray, T: np.ndarray, Y: np.ndarray):
        """Fit X-learner using two-stage approach."""
        X = np.asarray(X)
        T = np.asarray(T)
        Y = np.asarray(Y)

        treat_mask = (T == 1)
        control_mask = (T == 0)

        # Stage 1: Fit outcome models
        self.mu_0 = clone(self.outcome_model)
        self.mu_1 = clone(self.outcome_model)

        self.mu_0.fit(X[control_mask], Y[control_mask])
        self.mu_1.fit(X[treat_mask], Y[treat_mask])

        # Impute counterfactuals
        D_1 = Y[treat_mask] - self.mu_0.predict(X[treat_mask])
        D_0 = self.mu_1.predict(X[control_mask]) - Y[control_mask]

        # Stage 2: Fit effect models
        self.tau_0 = clone(self.effect_model)
        self.tau_1 = clone(self.effect_model)
```

```
        self.tau_0.fit(X[control_mask], D_0)
        self.tau_1.fit(X[treat_mask], D_1)

        # Propensity score for weighting
        if self.propensity_model is not None:
            self.propensity = clone(self.propensity_model)
            self.propensity.fit(X, T)

        return self

    def predict(self, X: np.ndarray) -> np.ndarray:
        """Predict CATE using propensity-weighted combination."""
        X = np.asarray(X)

        tau_0 = self.tau_0.predict(X)
        tau_1 = self.tau_1.predict(X)

        if self.propensity is not None:
            # Propensity-weighted combination
            e = self.propensity.predict_proba(X)[:, 1]
            return e * tau_0 + (1 - e) * tau_1
        else:
            # Simple average
            return 0.5 * (tau_0 + tau_1)


class CATEAnalyzer:
    """
    High-level CATE analysis with confidence intervals.
    """

    def __init__(self, learner_type: str = "t"):
        if learner_type == "t":
            self.learner = TLearner()
        elif learner_type == "s":
            self.learner = SLearner()
        elif learner_type == "x":
            self.learner = XLearner()
        else:
            raise ValueError(f"Unknown learner type: {learner_type}")

    def fit(self, X: np.ndarray, T: np.ndarray, Y: np.ndarray):
        """Fit CATE model."""
        self.learner.fit(X, T, Y)
        return self

    def predict(self, X: np.ndarray) -> np.ndarray:
        """Predict CATE."""
        return self.learner.predict(X)

    def predict_interval(
        self,
        X: np.ndarray,
        n_bootstrap: int = 100,
        alpha: float = 0.05
    ) -> tuple:
        """
        Predict CATE with bootstrap confidence intervals.
        """
        # This requires refitting, so store training data
        # Simplified version using point estimate variance
        cate = self.predict(X)
```

```
        # Bootstrap would go here
        # For simplicity, return wide intervals
        margin = 0.1 * np.abs(cate) + 0.01

        return cate, cate - margin, cate + margin
```

1.8 Implementation: Production Assignment Service

```
"""
Production experiment assignment with consistent hashing.
"""

import hashlib
from typing import Dict, Optional, List
from dataclasses import dataclass


@dataclass
class ExperimentConfig:
    """Configuration for an experiment."""
    experiment_id: str
    variants: List[str]
    allocation: Dict[str, float]  # variant -> proportion
    traffic: float  # fraction of users in experiment
    enabled: bool = True
    targeting: Optional[Dict] = None  # optional targeting rules


class AssignmentService:
    """
    Deterministic, consistent experiment assignment.
    Uses hashing for reproducible assignments.
    """

    def __init__(self):
        self.experiments: Dict[str, ExperimentConfig] = {}

    def register_experiment(self, config: ExperimentConfig):
        """Register an experiment configuration."""
        # Validate allocation sums to 1
        total_alloc = sum(config.allocation.values())
        if abs(total_alloc - 1.0) > 1e-6:
            raise ValueError(f"Allocation must sum to 1, got {total_alloc}")

        self.experiments[config.experiment_id] = config

    def _hash(self, salt: str, user_id: str) -> float:
        """Consistent hash to [0, 1]."""
        hash_input = f"{salt}:{user_id}".encode("utf-8")
        hash_bytes = hashlib.md5(hash_input).digest()
        hash_int = int.from_bytes(hash_bytes[:8], byteorder="big")
        return hash_int / (2**64)

    def assign(
        self,
        experiment_id: str,
        user_id: str,
        user_context: Optional[Dict] = None
```

```
    ) -> Optional[str]:
        """
        Assign user to variant.

        Returns variant name or None if not in experiment.
        """
        if experiment_id not in self.experiments:
            return None

        config = self.experiments[experiment_id]

        if not config.enabled:
            return None

        # Check targeting rules
        if config.targeting and user_context:
            if not self._check_targeting(config.targeting, user_context):
                return None

        # Traffic allocation check
        traffic_hash = self._hash(f"{experiment_id}:traffic", user_id)
        if traffic_hash >= config.traffic:
            return None

        # Variant assignment
        variant_hash = self._hash(f"{experiment_id}:variant", user_id)

        cumulative = 0.0
        for variant, proportion in config.allocation.items():
            cumulative += proportion
            if variant_hash < cumulative:
                return variant

        # Fallback to last variant
        return list(config.allocation.keys())[-1]

    def _check_targeting(self, rules: Dict, context: Dict) -> bool:
        """Check if user matches targeting rules."""
        for key, expected in rules.items():
            if key not in context:
                return False
            if isinstance(expected, list):
                if context[key] not in expected:
                    return False
            elif context[key] != expected:
                return False
        return True

    def get_assignment_distribution(
        self,
        experiment_id: str,
        n_users: int = 10000
    ) -> Dict[str, int]:
        """Simulate assignment distribution for verification."""
        counts = {}
        for i in range(n_users):
            variant = self.assign(experiment_id, f"user_{i}")
            if variant is not None:
                counts[variant] = counts.get(variant, 0) + 1
        return counts
```

1.9 Common Parameters

0.25	0.3409	Typical Range	

0.25	100000	De-	
pleper		pends	
size-	arm	on	
		base-	
		line	
		rate	
		and	
		MDE	

0.25	0.05	T-	
nif-	pha	type	
i-		er-	
cance		ror	
		rate	

0.25	0.95	Prob-	
	beta	a-	
		bil-	
		ity	
		of	
		de-	
		tect-	
		ing	
		true	
		ef-	
		fect	

0.25	0.10	Mini-	
		mum	
		de-	
		tectable	
		rel-	
		a-	
		tive	
		ef-	
		fect	

0.25	0.5	0.75	Typical
			0.3409 Range
0.25	0.5	0.75	Num-
quen-	10		ber
tial			of
			in-
			terim
			anal-
			y-
			ses
0.25	0.5	0.75	Pre-
re-	0.9		experiment
la-			co-
tion			vari-
			ate
			cor-
			re-
			la-
			tion
0.25	0.5	0.75	Num-
i-	20		ber
fi-			of
ca-			strat-
tion			i-
			fi-
			ca-
			tion
			groups
0.25	0.5	0.75	Frac-
sig-	1.0		tion
ment			of
			users
			in
			ex-
			per-
			i-
			ment

1.10 Practice Problems

1. **Power Analysis**: For a 5% baseline conversion rate and 10% relative MDE, calculate the required sample size per arm at 80% power and 5% significance level.

2. **CUPED Implementation**: Given pre-experiment purchase data with correlation 0.6 to experimental outcome, implement CUPED and verify the variance reduction factor.

3. **Sequential Testing**: Design an O'Brien-Fleming stopping rule for 4 interim analyses. Compute the boundaries and compare to Pocock boundaries.
4. **CATE Estimation**: Train a T-learner on heterogeneous treatment data. Identify subgroups with largest treatment effects.
5. **Network Experiments**: Design a cluster-randomized experiment for a social network with 100 communities. Account for within-cluster correlation in sample size calculation.

1.11 References

1. Imbens, G. W., & Rubin, D. B. (2015). "Causal Inference for Statistics, Social, and Biomedical Sciences." Cambridge University Press.
2. Kohavi, R., Tang, D., & Xu, Y. (2020). "Trustworthy Online Controlled Experiments: A Practical Guide to A/B Testing." Cambridge University Press.
3. Deng, A., et al. (2013). "Improving the Sensitivity of Online Controlled Experiments by Utilizing Pre-Experiment Data." WSDM.
4. Howard, S. R., et al. (2021). "Time-Uniform, Nonparametric, Nonasymptotic Confidence Sequences." Annals of Statistics.
5. Kunzel, S. R., et al. (2019). "Metalearners for Estimating Heterogeneous Treatment Effects Using Machine Learning." PNAS.
6. Athey, S., & Imbens, G. W. (2016). "Recursive Partitioning for Heterogeneous Causal Effects." PNAS.

A/B testing bridges statistical theory with product decisions. Rigorous experimental design and valid inference transform uncertainty into actionable insights.