Handout 2: Intermediate A/B Testing Experimentation

Machine Learning for Smarter Innovation

1 Handout 2: Intermediate A/B Testing & Experimentation

1.1 Week 10: A/B Testing & Iterative Improvement

Skill Level: Intermediate | Python & Statistics Required | For Data Scientists & ML Engineers

---

1.2 Prerequisites

Required Knowledge: - Python programming (numpy, pandas, scipy) - Basic statistics (hypothesis testing, confidence intervals) - Probability distributions (normal, binomial, beta)

Required Libraries:

```
import numpy as np
import pandas as pd
from scipy import stats
from scipy.stats import beta, binom
import statsmodels.stats.proportion as smp
import matplotlib.pyplot as plt
```

1.3 Part 1: Classical (Frequentist) A/B Testing

1.3.1 Sample Size Calculation

Before running any test, calculate the required sample size.

Formula:

```
n = (z_alpha/2 + z_beta)^2 * 2 * p * (1-p) / delta^2
```

Where: - z_alpha/2: Critical value for significance level (1.96 for 95% confidence) - z_beta: Critical value for power (0.84 for 80% power) - p: Baseline conversion rate - delta: Minimum detectable effect

Python Implementation:

```
from statsmodels.stats.power import zt_ind_solve_power

def calculate_sample_size(p1, p2, alpha=0.05, power=0.8):
```

```
    """
    Calculate required sample size per group for proportion test.

    Parameters:
    -----------
    p1 : float
        Baseline conversion rate (control)
    p2 : float
        Expected treatment conversion rate
    alpha : float
        Significance level (default 0.05)
    power : float
        Statistical power (default 0.8)

    Returns:
    --------
    n : int
        Required sample size per group
    """
    effect_size = (p2 - p1) / np.sqrt(p1 * (1 - p1))

    n = zt_ind_solve_power(
        effect_size=effect_size,
        alpha=alpha,
        power=power,
        ratio=1.0,
        alternative='two-sided'
    )

    return int(np.ceil(n))

# Example: Detect 1pp increase from 5% to 6%
baseline = 0.05
treatment = 0.06
n_required = calculate_sample_size(baseline, treatment)
print(f"Required sample size per group: {n_required}")
# Output: Required sample size per group: 8844
```

Interpretation: - Need 8,844 users per group (17,688 total) - Smaller effects require more users - Higher power requires more users

1.3.2 Z-Test for Proportions

Most common A/B test: comparing conversion rates between two groups.

Python Implementation:

```
def proportion_ztest(control, treatment, alpha=0.05):
    """
    Perform two-proportion z-test.

    Parameters:
    -----------
    control : array-like
        Binary outcomes for control group (0 or 1)
    treatment : array-like
        Binary outcomes for treatment group (0 or 1)
    alpha : float
        Significance level

    Returns:
    --------
```

```
    results : dict
        Test statistics, p-value, confidence interval
    """
    n_control = len(control)
    n_treatment = len(treatment)

    conversions_control = sum(control)
    conversions_treatment = sum(treatment)

    p_control = conversions_control / n_control
    p_treatment = conversions_treatment / n_treatment

    # Z-test using statsmodels
    count = np.array([conversions_treatment, conversions_control])
    nobs = np.array([n_treatment, n_control])

    z_stat, p_value = smp.proportions_ztest(count, nobs)

    # Confidence interval for difference
    diff = p_treatment - p_control
    se_diff = np.sqrt(
        p_control * (1 - p_control) / n_control +
        p_treatment * (1 - p_treatment) / n_treatment
    )
    ci_lower = diff - 1.96 * se_diff
    ci_upper = diff + 1.96 * se_diff

    # Relative lift
    lift = (p_treatment - p_control) / p_control if p_control > 0 else np.inf

    return {
        'p_control': p_control,
        'p_treatment': p_treatment,
        'absolute_diff': diff,
        'relative_lift': lift,
        'z_statistic': z_stat,
        'p_value': p_value,
        'ci_lower': ci_lower,
        'ci_upper': ci_upper,
        'significant': p_value < alpha
    }

# Example: Real A/B test data
np.random.seed(42)
control = np.random.binomial(1, 0.05, 10000)
treatment = np.random.binomial(1, 0.06, 10000)

results = proportion_ztest(control, treatment)

print(f"Control conversion: {results['p_control']:.2%}")
print(f"Treatment conversion: {results['p_treatment']:.2%}")
print(f"Absolute difference: {results['absolute_diff']:.2%}")
print(f"Relative lift: {results['relative_lift']:.1%}")
print(f"P-value: {results['p_value']:.4f}")
print(f"95% CI: [{results['ci_lower']:.2%}, {results['ci_upper']:.2%}]")
print(f"Significant: {results['significant']}")
```

Output:

```
Control conversion: 5.07%
Treatment conversion: 6.18%
Absolute difference: 1.11%
Relative lift: 21.9%
```

```
P-value: 0.0023
95% CI: [0.40%, 1.82%]
Significant: True
```

1.3.3 T-Test for Continuous Metrics

For continuous outcomes like revenue or time-on-site.

Python Implementation:

```python
def ttest_ab(control, treatment, alpha=0.05):
    """
    Perform two-sample t-test for continuous metrics.

    Parameters:
    -----------
    control : array-like
        Continuous outcomes for control group
    treatment : array-like
        Continuous outcomes for treatment group
    alpha : float
        Significance level

    Returns:
    --------
    results : dict
        Test statistics, p-value, confidence interval
    """
    mean_control = np.mean(control)
    mean_treatment = np.mean(treatment)

    # Two-sample t-test (Welch's t-test, unequal variances)
    t_stat, p_value = stats.ttest_ind(treatment, control, equal_var=False)

    # Confidence interval for difference
    n_control = len(control)
    n_treatment = len(treatment)

    var_control = np.var(control, ddof=1)
    var_treatment = np.var(treatment, ddof=1)

    se_diff = np.sqrt(var_control / n_control + var_treatment / n_treatment)
    diff = mean_treatment - mean_control

    # Degrees of freedom (Welch-Satterthwaite)
    df = (var_control / n_control + var_treatment / n_treatment)**2 / (
        (var_control / n_control)**2 / (n_control - 1) +
        (var_treatment / n_treatment)**2 / (n_treatment - 1)
    )

    t_crit = stats.t.ppf(1 - alpha/2, df)
    ci_lower = diff - t_crit * se_diff
    ci_upper = diff + t_crit * se_diff

    # Relative lift
    lift = diff / mean_control if mean_control > 0 else np.inf

    return {
        'mean_control': mean_control,
        'mean_treatment': mean_treatment,
        'absolute_diff': diff,
        'relative_lift': lift,
```

```
        't_statistic': t_stat,
        'p_value': p_value,
        'ci_lower': ci_lower,
        'ci_upper': ci_upper,
        'significant': p_value < alpha
    }

# Example: Revenue per user
np.random.seed(42)
control_revenue = np.random.gamma(2, 10, 10000)  # Mean $20
treatment_revenue = np.random.gamma(2.2, 10, 10000)  # Mean $22

results = ttest_ab(control_revenue, treatment_revenue)

print(f"Control mean: ${results['mean_control']:.2f}")
print(f"Treatment mean: ${results['mean_treatment']:.2f}")
print(f"Absolute difference: ${results['absolute_diff']:.2f}")
print(f"Relative lift: {results['relative_lift']:.1%}")
print(f"P-value: {results['p_value']:.4f}")
print(f"95% CI: [${results['ci_lower']:.2f}, ${results['ci_upper']:.2f}]")
print(f"Significant: {results['significant']}")
```

1.4 Part 2: Bayesian A/B Testing

Bayesian methods provide intuitive probability statements like "95% probability Treatment is better than Control."

1.4.1 Beta-Binomial Model for Conversion Rates

Theory: - Prior: Beta(alpha, beta) = Beta(1, 1) = Uniform(0, 1) - Likelihood: Binomial(n, p) - Posterior: Beta(alpha + conversions, beta + non_conversions)

Python Implementation:

```
from scipy.stats import beta

def bayesian_ab_test(control_conversions, control_visitors,
                     treatment_conversions, treatment_visitors,
                     prior_alpha=1, prior_beta=1, n_samples=100000):
    """
    Bayesian A/B test using Beta-Binomial conjugate prior.

    Parameters:
    -----------
    control_conversions : int
        Number of conversions in control
    control_visitors : int
        Number of visitors in control
    treatment_conversions : int
        Number of conversions in treatment
    treatment_visitors : int
        Number of visitors in treatment
    prior_alpha, prior_beta : float
        Beta prior parameters (default: uniform prior)
    n_samples : int
        Number of Monte Carlo samples

    Returns:
```

```
    --------
    results : dict
        Posterior statistics and probabilities
    """
    # Posterior parameters
    posterior_a_control = prior_alpha + control_conversions
    posterior_b_control = prior_beta + (control_visitors - control_conversions
    )

    posterior_a_treatment = prior_alpha + treatment_conversions
    posterior_b_treatment = prior_beta + (treatment_visitors -
    treatment_conversions)

    # Sample from posteriors
    samples_control = beta.rvs(posterior_a_control, posterior_b_control, size=
    n_samples)
    samples_treatment = beta.rvs(posterior_a_treatment, posterior_b_treatment,
     size=n_samples)

    # Probability treatment is better
    prob_treatment_better = (samples_treatment > samples_control).mean()

    # Expected loss
    loss_if_choose_control = np.maximum(samples_treatment - samples_control,
    0).mean()
    loss_if_choose_treatment = np.maximum(samples_control - samples_treatment,
     0).mean()

    # Relative lift distribution
    lift = (samples_treatment - samples_control) / samples_control

    return {
        'posterior_mean_control': posterior_a_control / (posterior_a_control +
     posterior_b_control),
        'posterior_mean_treatment': posterior_a_treatment / (
    posterior_a_treatment + posterior_b_treatment),
        'prob_treatment_better': prob_treatment_better,
        'expected_loss_control': loss_if_choose_control,
        'expected_loss_treatment': loss_if_choose_treatment,
        'lift_mean': lift.mean(),
        'lift_median': np.median(lift),
        'lift_95_ci': (np.percentile(lift, 2.5), np.percentile(lift, 97.5))
    }

# Example
control_conv = 500
control_vis = 10000
treatment_conv = 580
treatment_vis = 10000

results = bayesian_ab_test(control_conv, control_vis, treatment_conv,
    treatment_vis)

print(f"Control rate: {results['posterior_mean_control']:.2%}")
print(f"Treatment rate: {results['posterior_mean_treatment']:.2%}")
print(f"P(Treatment > Control): {results['prob_treatment_better']:.1%}")
print(f"Expected lift: {results['lift_mean']:.1%}")
print(f"95% Credible Interval for lift: [{results['lift_95_ci'][0]:.1%}, {
    results['lift_95_ci'][1]:.1%}]")
print(f"Expected loss if choose Control: {results['expected_loss_control']:.4f
    }")
print(f"Expected loss if choose Treatment: {results['expected_loss_treatment
    ']:.4f}")
```

Decision Rule: - If P(Treatment better than Control) greater than 95% AND expected loss less than 0.1%: Ship Treatment - If P(Control better than Treatment) greater than 95%: Stop - Otherwise: Continue collecting data

---

1.5 Part 3: Multi-Armed Bandits

Bandits balance exploration (learning which arms are best) with exploitation (using best arm).

1.5.1 Thompson Sampling

Algorithm: 1. Maintain Beta posterior for each arm 2. Sample from each posterior 3. Pull arm with highest sample 4. Update posterior with observed reward

Python Implementation:

```
class ThompsonSamplingBandit:
    """
    Thompson Sampling for multi-armed bandit with binary rewards.
    """
    def __init__(self, n_arms, prior_alpha=1, prior_beta=1):
        self.n_arms = n_arms
        self.alpha = np.full(n_arms, prior_alpha, dtype=float)
        self.beta = np.full(n_arms, prior_beta, dtype=float)
        self.total_pulls = np.zeros(n_arms)
        self.total_rewards = np.zeros(n_arms)

    def select_arm(self):
        """Sample from posteriors and return arm with highest sample."""
        samples = [np.random.beta(self.alpha[i], self.beta[i])
                   for i in range(self.n_arms)]
        return np.argmax(samples)

    def update(self, arm, reward):
        """Update posterior after observing reward."""
        self.alpha[arm] += reward
        self.beta[arm] += (1 - reward)
        self.total_pulls[arm] += 1
        self.total_rewards[arm] += reward

    def get_statistics(self):
        """Return posterior means and confidence intervals."""
        means = self.alpha / (self.alpha + self.beta)
        return {
            'means': means,
            'pulls': self.total_pulls,
            'rewards': self.total_rewards,
            'empirical_means': self.total_rewards / (self.total_pulls + 1e-8)
        }

# Simulation: Compare 3 recommendation algorithms
np.random.seed(42)
true_rates = [0.05, 0.07, 0.055]  # Algorithm 2 is best
n_rounds = 10000

bandit = ThompsonSamplingBandit(n_arms=3)

# Track regret
cumulative_regret = []
best_arm = np.argmax(true_rates)
```

```
for t in range(n_rounds):
    arm = bandit.select_arm()
    reward = np.random.binomial(1, true_rates[arm])
    bandit.update(arm, reward)

    # Calculate regret
    regret = true_rates[best_arm] - true_rates[arm]
    if t == 0:
        cumulative_regret.append(regret)
    else:
        cumulative_regret.append(cumulative_regret[-1] + regret)

stats = bandit.get_statistics()
print("\nThompson Sampling Results:")
for i in range(3):
    print(f"Algorithm {i+1}: {stats['pulls'][i]:.0f} pulls, "
          f"{stats['empirical_means'][i]:.2%} empirical rate")
print(f"\nTotal regret: {cumulative_regret[-1]:.1f}")
print(f"Best arm pulled: {stats['pulls'][best_arm]/n_rounds:.1%} of the time")
```

Output:

```
Thompson Sampling Results:
Algorithm 1: 1274 pulls, 5.02% empirical rate
Algorithm 2: 8142 pulls, 7.01% empirical rate
Algorithm 3: 584 pulls, 5.48% empirical rate

Total regret: 124.5
Best arm pulled: 81.4% of the time
```

Key Insight: Thompson Sampling quickly identifies Algorithm 2 as best and exploits it while still occasionally exploring others.

1.6 Part 4: Sequential Testing

Standard A/B tests require fixed sample size. Sequential testing allows early stopping while controlling error rates.

1.6.1 O'Brien-Fleming Boundaries

Theory: - Alpha spending function that makes early stopping very conservative - Late stopping approaches nominal alpha - Total alpha remains 0.05

Python Implementation:

```
def obrien_fleming_boundary(n_looks, alpha=0.05):
    """
    Calculate O'Brien-Fleming stopping boundaries.

    Parameters:
    -----------
    n_looks : int
        Number of planned interim analyses
    alpha : float
        Total type I error rate

    Returns:
```

```
    --------
    boundaries : array
        Z-score threshold for each look
    """
    looks = np.arange(1, n_looks + 1)
    boundaries = stats.norm.ppf(1 - alpha/2) * np.sqrt(n_looks / looks)
    return boundaries

# Example: Plan 5 interim looks
boundaries = obrien_fleming_boundary(n_looks=5)
print("O'Brien-Fleming boundaries:")
for i, b in enumerate(boundaries, 1):
    print(f"Look {i}: z > {b:.3f} (approx p < {2*(1-stats.norm.cdf(b)):.5f})")
```

Output:

```
O'Brien-Fleming boundaries:
Look 1: z > 4.381 (approx p < 0.00001)
Look 2: z > 3.098 (approx p < 0.00195)
Look 3: z > 2.529 (approx p < 0.01144)
Look 4: z > 2.191 (approx p < 0.02847)
Look 5: z > 1.960 (approx p < 0.05000)
```

Interpretation: - At first look (20% of data), need z greater than 4.38 to stop - At final look (100% of data), use standard z greater than 1.96 - Conservative early, liberal late

---

1.7 Part 5: Real-World Implementation

1.7.1 Complete A/B Testing Pipeline

```
class ABTestPipeline:
    """
    End-to-end A/B testing pipeline with pre-checks and post-analysis.
    """
    def __init__(self, alpha=0.05, power=0.8, mde=0.05):
        self.alpha = alpha
        self.power = power
        self.mde = mde

    def pre_test_checks(self, baseline_rate, expected_traffic):
        """Validate test is feasible."""
        n_required = calculate_sample_size(
            baseline_rate,
            baseline_rate * (1 + self.mde),
            self.alpha,
            self.power
        )

        duration_days = (2 * n_required) / expected_traffic

        return {
            'required_sample_size': n_required,
            'estimated_duration_days': duration_days,
            'feasible': duration_days <= 30
        }

    def run_test(self, control, treatment, test_type='proportion'):
        """Execute appropriate statistical test."""
```

```
        if test_type == 'proportion':
            return proportion_ztest(control, treatment, self.alpha)
        elif test_type == 'continuous':
            return ttest_ab(control, treatment, self.alpha)
        else:
            raise ValueError("test_type must be 'proportion' or 'continuous'")

    def segment_analysis(self, control, treatment, segments):
        """Analyze by segments to detect Simpson's paradox."""
        results = {}
        for segment_name in segments.unique():
            segment_mask = (segments == segment_name)
            control_seg = control[segment_mask]
            treatment_seg = treatment[segment_mask]

            results[segment_name] = proportion_ztest(control_seg,
    treatment_seg, self.alpha)

        return results

    def guardrail_check(self, guardrails):
        """Check if any guardrail metrics degraded significantly."""
        failures = []
        for metric_name, results in guardrails.items():
            if results['significant'] and results['absolute_diff'] < 0:
                failures.append(metric_name)
        return failures

    def make_decision(self, primary_results, guardrail_failures):
        """Generate final recommendation."""
        if guardrail_failures:
            return "ROLLBACK: Guardrail failures detected"

        if not primary_results['significant']:
            return "STOP: No significant difference detected"

        lift = primary_results['relative_lift']
        p_val = primary_results['p_value']

        if p_val < 0.01 and abs(lift) > 0.10:
            return "SHIP: Strong winner with high confidence"
        elif p_val < 0.05 and abs(lift) > 0.02:
            return "SHIP: Moderate winner, monitor closely"
        else:
            return "ITERATE: Effect too small for impact"

# Example usage
pipeline = ABTestPipeline(alpha=0.05, power=0.8, mde=0.05)

# Pre-test planning
pre_check = pipeline.pre_test_checks(baseline_rate=0.05, expected_traffic
    =1000)
print(f"Required sample size: {pre_check['required_sample_size']}")
print(f"Estimated duration: {pre_check['estimated_duration_days']:.1f} days")

# Generate test data
np.random.seed(42)
control = np.random.binomial(1, 0.05, 10000)
treatment = np.random.binomial(1, 0.06, 10000)

# Run test
results = pipeline.run_test(control, treatment, test_type='proportion')
```

```
# Check guardrails
guardrail_results = {
    'error_rate': proportion_ztest(
        np.random.binomial(1, 0.001, 10000),
        np.random.binomial(1, 0.002, 10000)
    ),
    'page_load_time': ttest_ab(
        np.random.normal(150, 20, 10000),
        np.random.normal(155, 20, 10000)
    )
}

guardrail_failures = pipeline.guardrail_check(guardrail_results)

# Decision
decision = pipeline.make_decision(results, guardrail_failures)
print(f"\nDecision: {decision}")
```

1.8 Part 6: Common Pitfalls and Solutions

1.8.1 Pitfall 1: Multiple Testing Problem

Problem: Running 20 A/B tests, 1 will show p less than 0.05 by chance.

Solution: Bonferroni correction

```
def bonferroni_correction(p_values, alpha=0.05):
    """Apply Bonferroni correction for multiple comparisons."""
    n_tests = len(p_values)
    adjusted_alpha = alpha / n_tests
    return [p < adjusted_alpha for p in p_values]

# Example: 3 tests
p_values = [0.02, 0.04, 0.06]
significant = bonferroni_correction(p_values)
print(f"Adjusted threshold: {0.05/3:.4f}")
print(f"Significant after correction: {significant}")
# Output: [True, False, False]
```

1.8.2 Pitfall 2: Non-Stationarity

Problem: User behavior changes over time (day of week, holidays).

Solution: Stratified sampling and time-based analysis

```
def stratified_test(control, treatment, time_buckets):
    """
    Test accounting for time-based non-stationarity.

    Parameters:
    -----------
    control, treatment : array-like
        Outcomes for each group
    time_buckets : array-like
        Time period identifier for each observation

    Returns:
    --------
```

```
    Overall and per-bucket results
    """
    overall = proportion_ztest(control, treatment)

    bucket_results = {}
    for bucket in np.unique(time_buckets):
        mask = (time_buckets == bucket)
        bucket_results[bucket] = proportion_ztest(control[mask], treatment[
mask])

    return {
        'overall': overall,
        'by_bucket': bucket_results
    }
```

1.9 Key Takeaways

1. **Always calculate sample size before running test**
   - Prevents underpowered experiments
   - Sets clear stopping criteria
2. **Use both frequentist and Bayesian methods**
   - Frequentist: Industry standard, regulatory requirements
   - Bayesian: Intuitive probabilities, early stopping
3. **Multi-armed bandits optimize during learning**
   - Thompson Sampling balances exploration and exploitation
   - Reduces opportunity cost compared to fixed A/B tests
4. **Sequential testing enables early stopping**
   - O'Brien-Fleming boundaries control type I error
   - Can stop winners early, reducing experiment duration
5. **Always check guardrails and segments**
   - Prevent shipping winners that break something else
   - Detect Simpson's paradox before making wrong decision

Next Steps: - Read Handout 3 for advanced causal inference methods - Implement these functions in your experimentation platform - Run your first Bayesian A/B test this week