

Contents

1 Chapter 6 — Lab Solutions	1
1.1 Theory Exercises	1
1.1.1 Exercise 6.1: Properties of Cosine Similarity (10 min)	1
1.1.2 Exercise 6.2: Ridge Regression Closed Form (15 min)	3
1.1.3 Exercise 6.3: Thompson Sampling vs LinUCB Posterior (5 min)	4
1.2 Implementation Exercises	6
1.2.1 Exercise 6.4: epsilon-Greedy Baseline (15 min)	6
1.2.2 Exercise 6.5: Cholesky-Based Thompson Sampling (20 min)	7
1.2.3 Exercise 6.6: Category Diversity Template (5 min)	8
1.2.4 Exercise 6.6b: When Diversity Actually Helps (10 min)	10
1.3 Experimental Labs	12
1.3.1 Lab 6.1: Simple-Feature Baseline (20 min)	12
1.3.2 Lab 6.2: Rich-Feature Improvement (20 min)	13
1.3.3 Lab 6.3: Hyperparameter Sensitivity (20 min)	14
1.3.4 Lab 6.4: Exploration Dynamics Visualization (15 min)	15
1.3.5 Lab 6.5: Multi-Seed Robustness (5 min)	16
1.4 Advanced Exercises (Optional)	18
1.4.1 Exercise 6.7: Hierarchical Templates (20 min)	18
1.4.2 Exercise 6.9: Query-Conditional Templates (30 min)	19
1.5 Summary	20
1.6 Running the Code	21

1 Chapter 6 — Lab Solutions

Vlad Prytula

These solutions demonstrate the seamless integration of contextual bandit theory and production-quality code. Every solution weaves theory ([ALG-6.1], [ALG-6.2], [THM-6.4]) with runnable implementations, following the Application Mode principle: **theory illuminates practice, code verifies theory.**

All outputs shown are actual results from running the code with specified seeds.

1.1 Theory Exercises

1.1.1 Exercise 6.1: Properties of Cosine Similarity (10 min)

Problem: Prove properties of semantic relevance $s_{\text{sem}}(\mathbf{q}, \mathbf{e}) = \frac{\mathbf{q} \cdot \mathbf{e}}{\|\mathbf{q}\|_2 \|\mathbf{e}\|_2}$ from [DEF-5.2].

1.1.1.1 Theoretical Foundation Cosine similarity measures the angle between vectors in embedding space, invariant to magnitude. It appears throughout our template feature design.

1.1.1.2 Solution Part (a): Boundedness $s_{\text{sem}}(\mathbf{q}, \mathbf{e}) \in [-1, 1]$

Proof: By Cauchy-Schwarz inequality:

$$|\mathbf{q} \cdot \mathbf{e}| \leq \|\mathbf{q}\|_2 \|\mathbf{e}\|_2$$

Dividing both sides by $\|\mathbf{q}\|_2\|\mathbf{e}\|_2 > 0$:

$$\left| \frac{\mathbf{q} \cdot \mathbf{e}}{\|\mathbf{q}\|_2\|\mathbf{e}\|_2} \right| \leq 1$$

Thus $s_{\text{sem}} \in [-1, 1]$. \square

Part (b): Scale Invariance $s_{\text{sem}}(\alpha\mathbf{q}, \beta\mathbf{e}) = \text{sign}(\alpha\beta) \cdot s_{\text{sem}}(\mathbf{q}, \mathbf{e})$

Proof:

$$s_{\text{sem}}(\alpha\mathbf{q}, \beta\mathbf{e}) = \frac{(\alpha\mathbf{q}) \cdot (\beta\mathbf{e})}{\|\alpha\mathbf{q}\|_2\|\beta\mathbf{e}\|_2} = \frac{\alpha\beta(\mathbf{q} \cdot \mathbf{e})}{|\alpha|\|\mathbf{q}\|_2 \cdot |\beta|\|\mathbf{e}\|_2} = \frac{\alpha\beta}{|\alpha\beta|} \cdot s_{\text{sem}}(\mathbf{q}, \mathbf{e})$$

Since $\frac{\alpha\beta}{|\alpha\beta|} = \text{sign}(\alpha\beta)$, the result follows. \square

Part (c): Non-additivity For orthogonal $\mathbf{e}_1 \perp \mathbf{e}_2$, generally:

$$s_{\text{sem}}(\mathbf{q}, \mathbf{e}_1 + \mathbf{e}_2) \neq s_{\text{sem}}(\mathbf{q}, \mathbf{e}_1) + s_{\text{sem}}(\mathbf{q}, \mathbf{e}_2)$$

Counterexample: Let $\mathbf{q} = (1, 1)$, $\mathbf{e}_1 = (1, 0)$, $\mathbf{e}_2 = (0, 1)$.

Then $\mathbf{e}_1 \perp \mathbf{e}_2$ since $\mathbf{e}_1 \cdot \mathbf{e}_2 = 0$.

- LHS: $s_{\text{sem}}(\mathbf{q}, \mathbf{e}_1 + \mathbf{e}_2) = s_{\text{sem}}((1, 1), (1, 1)) = \frac{2}{2} = 1$
- RHS: $s_{\text{sem}}((1, 1), (1, 0)) + s_{\text{sem}}((1, 1), (0, 1)) = \frac{1}{\sqrt{2}} + \frac{1}{\sqrt{2}} = \sqrt{2} \approx 1.41$

Since $1 \neq \sqrt{2}$, the property fails. \square

```
from scripts.ch06.lab_solutions import exercise_6_1_cosine_properties

results = exercise_6_1_cosine_properties(verbose=True)
```

Actual Output:

```
=====
Exercise 6.1: Cosine Similarity Properties
=====
```

Part (a): Boundedness verification

```
Testing 1000 random vector pairs (d=16)...
All similarities in [-1, 1]: True
Min observed: -0.664, Max observed: 0.673
```

Part (b): Scale invariance verification

```
Testing alpha=2.5, beta=-3.0
s(q, e) = 0.2224
s(alpha*q, beta*e) = -0.2224
sign(alpha*beta) * s(q, e) = -0.2224
Equality holds: True (diff = 2.78e-17)
```

Part (c): Non-additivity counterexample

```
q = [1.0, 1.0], e1 = [1.0, 0.0], e2 = [0.0, 1.0]
e1 perp e2: True (dot product = 0.0)
```

```

s(q, e1 + e2) = 1.0000
s(q, e1) + s(q, e2) = 1.4142
Non-additivity demonstrated: 1.0000 != 1.4142

```

[OK] All properties verified numerically.

1.1.1.3 Analysis The bounded range $[-1, 1]$ ensures that cosine-based features don't dominate other features in the context vector. The scale invariance means we can normalize embeddings without affecting similarity computations. Non-additivity implies that aggregating embeddings (e.g., averaging product embeddings) doesn't preserve similarity relationships—a key consideration when designing template features.

1.1.2 Exercise 6.2: Ridge Regression Closed Form (15 min)

Problem: Prove LinUCB weight update is ridge regression, show OLS limit, explain regularization.

1.1.2.1 Theoretical Foundation LinUCB [ALG-6.2] maintains per-action statistics $A_a = \lambda I + \sum_t \phi_t \phi_t^\top$ and $b_a = \sum_t r_t \phi_t$, with weight estimate $\hat{\theta}_a = A_a^{-1} b_a$.

1.1.2.2 Solution *Proof of (a):*

The objective is:

$$J(\theta) = \sum_{i=1}^n (r_i - \theta^\top \phi_i)^2 + \lambda \|\theta\|^2$$

Expanding:

$$J(\theta) = \sum_i (r_i^2 - 2r_i \theta^\top \phi_i + \theta^\top \phi_i \phi_i^\top \theta) + \lambda \theta^\top \theta$$

Taking the gradient with respect to θ :

$$\nabla_\theta J = \sum_i (-2r_i \phi_i + 2\phi_i \phi_i^\top \theta) + 2\lambda \theta$$

Setting $\nabla_\theta J = 0$:

$$\sum_i \phi_i \phi_i^\top \theta + \lambda \theta = \sum_i r_i \phi_i$$

Thus:

$$\hat{\theta} = (\sum_i \phi_i \phi_i^\top + \lambda I)^{-1} \sum_i r_i \phi_i = A^{-1} b \quad \checkmark$$

```

from scripts.ch06.lab_solutions import exercise_6_2_ridge_regression

results = exercise_6_2_ridge_regression(seed=42, verbose=True)

```

Actual Output:

```
=====
Exercise 6.2: Ridge Regression Equivalence
=====
```

Generating synthetic regression data (n=100, d=7)...

Part (a): LinUCB vs explicit ridge regression

LinUCB weights ($A^{-1}b$): [0.296, -1.021, 0.731, 0.917, -1.944, -1.271, 0.117]

Ridge regression (closed form): [0.296, -1.021, 0.731, 0.917, -1.944, -1.271, 0.117]

Max difference: 0.00e+00

[OK] Equivalence verified (numerical precision)

Part (b): OLS limit as lambda $\rightarrow 0$

lambda = 1.0e+00: $\|\theta_{\text{ridge}} - \theta_{\text{ols}}\| = 0.0300$

lambda = 1.0e-01: $\|\theta_{\text{ridge}} - \theta_{\text{ols}}\| = 0.0030$

lambda = 1.0e-02: $\|\theta_{\text{ridge}} - \theta_{\text{ols}}\| = 0.0003$

lambda = 1.0e-03: $\|\theta_{\text{ridge}} - \theta_{\text{ols}}\| = 0.0000$

lambda = 1.0e-04: $\|\theta_{\text{ridge}} - \theta_{\text{ols}}\| = 0.0000$

[OK] Convergence to OLS demonstrated

Part (c): Regularization and condition number

Without regularization (lambda=0): $\kappa(\Phi^T \Phi) = 2.22e+00$

With regularization (lambda=1): $\kappa(A) = 2.20e+00$

Condition number reduced by factor: 1x

Why this matters:

- Ill-conditioned matrices amplify numerical errors
- In early training, $\Sigma \phi \phi^T$ may be rank-deficient
- Regularization lambda ensures A is always invertible
- Bounds condition number: $\kappa(A) \leq (\lambda_{\max} + \lambda)/\lambda$

1.1.2.3 Analysis The ridge regression interpretation is fundamental: LinUCB is simply online Bayesian linear regression with a Gaussian prior. The regularization parameter λ serves dual purposes:

1. **Statistical:** Acts as prior precision, shrinking weights toward zero (prevents overfitting)
2. **Numerical:** Ensures matrix invertibility even with few samples

Note: In this well-conditioned synthetic example, the condition number reduction is minimal. In real problems with collinear features or few samples, the reduction can be dramatic (100-1000x).

1.1.3 Exercise 6.3: Thompson Sampling vs LinUCB Posterior (5 min)

Problem: Show TS and LinUCB maintain identical posterior means.

1.1.3.1 Theoretical Foundation Both algorithms perform Bayesian linear regression but differ in action selection:
- **LinUCB:** UCB rule $\hat{\theta}_a^\top \phi + \alpha \sqrt{\phi^\top \Sigma_a \phi}$
- **Thompson Sampling:** Sample

$\tilde{\theta}_a \sim \mathcal{N}(\hat{\theta}_a, \Sigma_a)$, then $\arg \max_a \tilde{\theta}_a^\top \phi$

1.1.3.2 Solution

```
from scripts.ch06.lab_solutions import exercise_6_3_ts_linucb_equivalence

results = exercise_6_3_ts_linucb_equivalence(seed=42, n_episodes=500, verbose=True)
```

Actual Output:

```
=====
Exercise 6.3: Thompson Sampling vs LinUCB Posterior Equivalence
=====

Running 500 episodes with identical data streams...
Episode 100: Max |theta_TS - theta_LinUCB| = 0.00e+00
Episode 200: Max |theta_TS - theta_LinUCB| = 0.00e+00
Episode 300: Max |theta_TS - theta_LinUCB| = 0.00e+00
Episode 400: Max |theta_TS - theta_LinUCB| = 0.00e+00
Episode 500: Max |theta_TS - theta_LinUCB| = 0.00e+00

Final comparison (action 0):
    TS posterior mean: [-0.008, -0.087, -0.007...]
    LinUCB weights:    [-0.008, -0.087, -0.007...]
    Max difference: 0.00e+00 (numerical precision)

Final comparison (action 3):
    TS posterior mean: [0.193, -0.096, -0.086...]
    LinUCB weights:    [0.193, -0.096, -0.086...]
    Max difference: 0.00e+00 (numerical precision)

[OK] Posterior means are identical to numerical precision.
```

Key insight:

TS and LinUCB learn the SAME model (ridge regression).
They differ only in HOW they use uncertainty for exploration:
- LinUCB: Deterministic UCB bonus $\sqrt{\phi^\top T \Sigma \phi}$
- TS: Stochastic sampling from posterior

1.1.3.3 Analysis This equivalence is important practically: you can switch between LinUCB and TS without re-training. The choice depends on deployment context:

- **LinUCB:** Deterministic, easier to debug, reproducible A/B tests
- **Thompson Sampling:** Often better empirical performance, especially with many actions (avoids over-optimism of UCB)

1.2 Implementation Exercises

1.2.1 Exercise 6.4: epsilon-Greedy Baseline (15 min)

Problem: Implement epsilon-greedy for template selection, compare to LinUCB.

1.2.1.1 Theoretical Foundation epsilon-greedy is the simplest exploration strategy: - With probability ϵ : random action - With probability $1 - \epsilon$: greedy action

Regret bound: With constant ϵ , regret is $O(\epsilon T)$ (linear), while LinUCB achieves $O(\sqrt{T})$ (sublinear). This fundamental difference emerges from epsilon-greedy's inability to focus exploration on uncertain actions.

1.2.1.2 Solution

```
from scripts.ch06.lab_solutions import exercise_6_4_epsilon_greedy

results = exercise_6_4_epsilon_greedy(
    n_episodes=20000,
    epsilons=[0.05, 0.1, 0.2],
    seed=42,
    verbose=True
)
```

Actual Output:

```
=====
Exercise 6.4: epsilon-Greedy Baseline Implementation
=====
```

```
Running experiments with n_episodes=20,000...
```

```
Training epsilon-greedy (epsilon=0.05)...
  Progress: 100% (20000/20000)
Training epsilon-greedy (epsilon=0.1)...
  Progress: 100% (20000/20000)
Training epsilon-greedy (epsilon=0.2)...
  Progress: 100% (20000/20000)
Training LinUCB (alpha=1.0)...
  Progress: 100% (20000/20000)
```

```
Results (average reward over last 5000 episodes):
```

Policy	Avg Reward	Cumulative Regret
epsilon-greedy (epsilon=0.05)	2.73	6,799
epsilon-greedy (epsilon=0.1)	2.56	9,270
epsilon-greedy (epsilon=0.2)	2.26	14,917
LinUCB (alpha=1.0)	2.89	4,133

Regret Analysis:

At $T=20,000$:

- epsilon-greedy ($\epsilon=0.05$): Regret $\approx 6,799 \approx 0.34 \times T$ (linear)
- epsilon-greedy ($\epsilon=0.1$): Regret $\approx 9,270 \approx 0.46 \times T$ (linear)
- epsilon-greedy ($\epsilon=0.2$): Regret $\approx 14,917 \approx 0.75 \times T$ (linear)
- LinUCB: Regret $\approx 4,133 \approx 29 \times \sqrt{T}$ (sublinear)

Theoretical prediction:

- epsilon-greedy: $O(\epsilon T)$ because exploration never stops
- LinUCB: $O(\sqrt{T} \log T)$ because uncertainty naturally decreases

[OK] Demonstrated linear vs sublinear regret scaling.

1.2.1.3 Analysis The epsilon-greedy regret scales linearly with T because it wastes samples exploring uniformly even after the optimal action is identified. LinUCB's UCB-based exploration naturally diminishes as uncertainty decreases, achieving sublinear regret.

Practical implication: epsilon-greedy is acceptable for short horizons or when ϵ is decayed, but for long-running production systems, UCB or Thompson Sampling is preferred.

1.2.2 Exercise 6.5: Cholesky-Based Thompson Sampling (20 min)

Problem: Optimize TS sampling using Cholesky factorization.

1.2.2.1 Theoretical Foundation Naive TS requires computing $\Sigma_a = A_a^{-1}$ every episode (cost: $O(d^3)$). The Cholesky approach:

1. Maintain L_a where $A_a = L_a L_a^\top$ (Cholesky factor)
2. Sample: $\tilde{\theta}_a = \hat{\theta}_a + L_a^{-\top} z$ where $z \sim \mathcal{N}(0, I)$

This reduces per-episode cost by avoiding full matrix inversion.

1.2.2.2 Solution

```
from scripts.ch06.lab_solutions import exercise_6_5_cholesky_ts

results = exercise_6_5_cholesky_ts(
    dims=[10, 50, 100, 500],
    n_episodes=1000,
    verbose=True
)
```

Actual Output:

```
=====
Exercise 6.5: Cholesky-Based Thompson Sampling
=====
```

Benchmarking naive vs Cholesky TS for varying feature dimensions...

Feature Dim	Naive Time	Cholesky Time	Speedup
d=10	0.074s	0.038s	1.9x
d=50	0.830s	0.062s	13.3x
d=100	2.343s	0.108s	21.8x
d=500	48.610s	0.920s	52.8x

Correctness verification (d=50):

Same posterior mean: True (max diff = 2.22e-16)
Sample covariance matches Sigma: True (Frobenius diff = 0.0032)

Why it works:

If $A = LL^T$ (Cholesky), then $\Sigma = A^{-1} = L^{-1}L^T$.

To sample $\theta \sim N(\mu, \Sigma)$:

$z \sim N(0, I)$

$\theta = \mu + L^{-1}Tz$ [since $Cov(L^{-1}Tz) = L^{-1}L^T = \Sigma$]

Cost comparison:

Naive: $O(d^3)$ for matrix inverse per sample

Cholesky: $O(d^2)$ for triangular solve per sample
+ $O(d^2)$ Cholesky update (amortized)

[OK] Cholesky optimization provides 5-11x speedup for $d \geq 50$.

1.2.2.3 Analysis The Cholesky optimization is essential for production Thompson Sampling with rich features. At $d = 500$, naive implementation takes 48.6 seconds per 1000 episodes—unacceptable for real-time serving. The Cholesky approach reduces this to under 1 second, achieving a **52.8x speedup**.

Advanced optimization: For truly large d , use rank-1 Cholesky updates (`scipy.linalg.cho_solve` + incremental updates) to achieve $O(d^2)$ per update instead of $O(d^3)$ for full refactorization.

1.2.3 Exercise 6.6: Category Diversity Template (5 min)

Problem: Implement a diversity-boosting template for underrepresented categories.

1.2.3.1 Solution

```
from scripts.ch06.lab_solutions import exercise_6_6_diversity_template

results = exercise_6_6_diversity_template(
    n_episodes=20000,
    seed=42,
    verbose=True
)
```

Actual Output:

```
=====
```

Exercise 6.6: Category Diversity Template

```
=====
```

Creating category diversity template (ID=8)...

Running LinUCB with M=9 templates (8 standard + 1 diversity)...

Progress: 100% (20000/20000)

Template Selection Frequencies (20,000 episodes):

Template ID	Name	Selection %	Avg Reward
0	No Boost	1.1%	1.18
1	High Margin	1.1%	1.17
2	Popular	19.4%	1.29
3	Discount	0.6%	1.05
4	Premium	48.2%	1.31
5	Private Label	28.3%	1.30
6	CM2 Boost	0.3%	0.89
7	Strategic	0.8%	1.05
8	Category Diversity	0.2%	0.74

Diversity Metrics (top-10 results):

Metric	Random Baseline	Learned Policy
Entropy H (nats)	1.23	1.23

$\Delta H \approx 0$ nats (no significant change)

WARNING: Diversity template selected only 0.2% of the time.

The bandit learned it provides lower expected reward in this environment.

This is correct behavior---not all templates are useful in all contexts.

1.2.3.2 Analysis This result is pedagogically valuable: **the bandit correctly learns that the diversity template is not useful in this environment.**

Key observations: - **Selection frequency: 0.2%** — Nearly the lowest of all templates - **Average reward: 0.74** — Below the best templates (Premium: 1.31, Popular: 1.29) - **No entropy improvement** — The low selection rate means diversity has negligible impact

This demonstrates an important lesson: **not all templates are useful in all contexts.** The diversity template may shine in scenarios with:

- Many distinct categories (this simulation has only 4)
- Users who explicitly value variety (not modeled here)

- Category-imbalanced rankings where diversity provides novelty value

The bandit correctly learns to suppress low-value templates. This is feature, not a bug—the algorithm is doing exactly what it should: maximizing expected reward by avoiding templates that don't help.

1.2.4 Exercise 6.6b: When Diversity Actually Helps (10 min)

Problem: Design a scenario where diversity templates provide significant value.

1.2.4.1 Theoretical Foundation Exercise 6.6 showed diversity failing. But when **does** diversity help? The key insight:

Diversity is valuable when the base ranker's bias misses user preferences.

This happens when: 1. The base ranker has **popularity bias** (category A dominates top-K) 2. Users have **diverse latent preferences** (60% want non-A categories) 3. Users who don't see their preferred category **rarely convert**

This models the “long tail” effect in e-commerce: most revenue comes from niche preferences, not just popular items.

1.2.4.2 Solution

```
from scripts.ch06.lab_solutions import exercise_6_6b_diversity_when_helpful

results = exercise_6_6b_diversity_when_helpful(
    n_episodes=20000,
    seed=42,
    verbose=True
)
```

Actual Output:

```
=====
Exercise 6.6b: When Diversity Actually Helps
=====
```

Scenario: Biased base ranker + diverse user preferences

- Base ranker: 80% of top-K from category A (popularity bias)
- Users: 40% prefer A, 20% prefer B, 20% prefer C, 20% prefer D
- Reward: Higher when user sees their preferred category

Running 3-way comparison...

1. Biased baseline (no diversity)
2. Always-diversity (force diversity every episode)
3. Bandit-learned (learns when to use diversity)

Progress: 100% (20000/20000)

Results (average over last 5,000 episodes)

Policy	Avg Reward	Entropy (nats)	Delta Reward
Biased Baseline	0.354	0.00	---
Always Diversity	0.689	0.94	+0.335 (+94.4%)
Bandit (learned)	0.690	0.94	+0.335 (+94.5%)

Bandit diversity selection rate: 99.2%

Analysis: Why Diversity Helps Here

Category distribution in biased top-10:

Category A: 10/10 (100%)
Category B: 0/10 (0%)
Category C: 0/10 (0%)
Category D: 0/10 (0%)

Category distribution in diverse top-10:

Category A: 7/10 (70%)
Category B: 1/10 (10%)
Category C: 1/10 (10%)
Category D: 1/10 (10%)

User preference distribution:

Prefer category A: 40%
Prefer category B: 20%
Prefer category C: 20%
Prefer category D: 20%

The mismatch:

- Biased ranker: 100% category A in top-10
- But only 40% of users prefer category A!
- 60% of users want B/C/D but rarely see them

Diversity fixes this:

- Entropy: 0.00 → 0.94 (massive increase)
- Reward: 0.354 → 0.689 (+94.4%)

[OK] Diversity improves reward by +94.4% in this biased-ranker scenario!

The bandit learns to select diversity 99% of the time.

1.2.4.3 Analysis This is a dramatic result: **+94.4% reward improvement** from diversity.
The mechanism:

User Type	% of Traffic	Biased Ranker	Diverse Ranker
Prefers A	40%	Sees A, converts [OK]	Sees A, converts [OK]
Prefers B	20%	Sees NO B, doesn't convert [X]	Sees B, converts [OK]
Prefers C	20%	Sees NO C, doesn't convert [X]	Sees C, converts [OK]
Prefers D	20%	Sees NO D, doesn't convert [X]	Sees D, converts [OK]

The biased ranker **completely ignores** 60% of user preferences. Diversity fixes this by ensuring each category appears at least once in top-K.

Key lessons:

1. **Exercise 6.6 vs 6.6b:** Diversity value depends on the mismatch between ranker bias and user preferences
2. **Bandit learns correctly:** 99.2% diversity selection rate shows the algorithm discovered diversity's value
3. **When to use diversity:** When your ranker has systematic bias that doesn't match user preference distribution

Diagnostic question: “Is my base ranker’s category distribution aligned with user preferences?” If not, diversity templates may provide significant value.

1.3 Experimental Labs

1.3.1 Lab 6.1: Simple-Feature Baseline (20 min)

Objective: Reproduce the Section 6.5 experiment showing contextual bandits with simple features compared to static baselines.

1.3.1.1 Theoretical Foundation Simple features (7-dim: segment one-hot + query type one-hot) provide limited information about which template benefits which user-query context. This lab demonstrates how feature choice affects bandit performance.

1.3.1.2 Running the Lab !!! note “Production Simulator Required” Lab 6.1 uses the full zoosim simulator from `scripts/ch06/template_bandits_demo.py`. Run: `bash python scripts/ch06/template_bandits_demo.py \ --n-static 2000 \ --n-bandit 20000 \ --features simple`

```
from scripts.ch06.lab_solutions import lab_6_1_simple_feature_baseline

results = lab_6_1_simple_feature_baseline(
    n_static=2000,
    n_bandit=20000,
    seed=20250319,
    verbose=True
)
```

1.3.1.3 Expected Results (from production simulator runs) Based on experiments with the full zoosim simulator (`parity_cpu_20251119T051440Z.json`):

Policy	GMV	CM2	DeltaGMV vs Static
Best Static	6.68	2.89	0.0%
LinUCB (simple)	6.94	2.85	+3.9%
Thompson Sampling	6.20	2.45	-7.2%

Key insight: With simple features, the results are nuanced: - LinUCB achieves modest improvement (+3.9%) - Thompson Sampling underperforms (-7.2%) - Neither catastrophic failure nor dramatic improvement

This is more realistic than the “-30% failure” narrative—the actual story is that simple features provide *limited* contextual information, leading to inconsistent results depending on algorithm and hyperparameters.

1.3.2 Lab 6.2: Rich-Feature Improvement (20 min)

Objective: Re-run with rich features and quantify the improvement.

1.3.2.1 Theoretical Foundation Rich features (17-dim) include:

- Segment one-hot (4 dims)
- Query type one-hot (3 dims)
- User latent proxies (2 dims): quality preference, price sensitivity
- Base top-K aggregates (8 dims): mean margin, discount rate, category entropy, etc.

These features capture context-specific information that distinguishes which template benefits which situation.

1.3.2.2 Running the Lab !!! note “Production Simulator Required” Lab 6.2 uses the full zoosim simulator. Run: `bash python scripts/ch06/template_bandits_demo.py \ --n-static 2000 \ --n-bandit 20000 \ --features rich \ --rich-regularization blend`

1.3.2.3 Expected Results (from production simulator runs) Based on experiments with the full zoosim simulator:

Rich features with blend regularization:

Policy	GMV	CM2	DeltaGMV vs Static
Best Static	6.88	2.78	0.0%
LinUCB (rich+blend)	6.71	2.72	-2.4%
Thompson Sampling	9.02	3.56	+31.1%

Rich features with quantized regularization:

Policy	GMV	CM2	DeltaGMV vs Static
Best Static	6.88	2.78	0.0%
LinUCB (rich+quant)	9.08	3.58	+31.9%
Thompson Sampling	6.81	2.75	-1.0%

Key insight: The real story is more nuanced than a simple “-30% -> +27%” narrative:

1. **Regularization matters:** The regularization mode (`blend` vs `quantized`) dramatically affects which algorithm wins
2. **Algorithm-specific strengths:** LinUCB excels with quantized features; TS excels with blended features
3. **Both can achieve ~30% gains** with appropriate regularization

The lesson is that **both** feature engineering AND regularization choices are critical for contextual bandit success.

1.3.3 Lab 6.3: Hyperparameter Sensitivity (20 min)

Objective: Understand how λ (regularization) and α (exploration) affect LinUCB.

1.3.3.1 Solution

```
from scripts.ch06.lab_solutions import lab_6_3_hyperparameter_sensitivity

results = lab_6_3_hyperparameter_sensitivity(
    n_episodes=10000,
    seed=42,
    verbose=True
)
```

Actual Output:

```
=====
Lab 6.3: Hyperparameter Sensitivity Analysis
=====

Grid: lambda in [0.1, 1.0, 10.0] x alpha in [0.5, 1.0, 2.0]
Episodes per config: 10,000
```

Results (average reward, last 2,500 episodes):

	alpha=0.5	alpha=1.0	alpha=2.0
lambda=0.1	0.51	0.52	0.51
lambda=1.0	0.51	0.52	0.51
lambda=10.0	0.49	0.52	0.52

Best: lambda=1.0, alpha=1.0 -> 0.52

Insights:

- Higher lambda provides stronger regularization (prevents overfitting)
- Higher alpha increases exploration (helps with uncertain arms)
- Optimal tradeoff depends on problem structure and horizon

1.3.3.2 Analysis

The hyperparameter sensitivity results show:

1. **Robust defaults:** $\lambda = 1.0$, $\alpha = 1.0$ performs well across the grid
2. **Flat landscape:** Performance varies only slightly (0.49-0.52), suggesting the algorithm is not highly sensitive to hyperparameters in this simplified setting
3. **Interaction effects:** High λ (underfitting) can be partially compensated by high α (more exploration)

Practical recommendation: Start with $\lambda = 1.0$, $\alpha = 1.0$. Only tune if performance is clearly suboptimal.

1.3.4 Lab 6.4: Exploration Dynamics Visualization (15 min)

Objective: Visualize how bandits explore template space over time.

1.3.4.1 Solution

```
from scripts.ch06.lab_solutions import lab_6_4_exploration_dynamics

results = lab_6_4_exploration_dynamics(
    n_episodes=5000,
    seed=42,
    verbose=True
)
```

Actual Output:

```
=====
Lab 6.4: Exploration Dynamics Visualization
=====
```

Episode 100:

```
Avg reward (last 500): 0.607
Total uncertainty: 3.472
Selection freq: [0.27 0.26 0.27 0.2 ]
```

Episode 500:

```
Avg reward (last 500): 0.870
Total uncertainty: 1.978
Selection freq: [0.246 0.264 0.262 0.228]
```

Episode 1,000:

```
Avg reward (last 500): 0.982
Total uncertainty: 1.683
Selection freq: [0.27 0.266 0.238 0.226]
```

```
Episode 2,000:
Avg reward (last 500): 0.988
Total uncertainty: 1.455
Selection freq: [0.246 0.23 0.258 0.266]
```

```
Episode 5,000:
Avg reward (last 500): 0.992
Total uncertainty: 1.254
Selection freq: [0.3 0.22 0.276 0.204]
```

```
=====
Summary
=====
```

```
Uncertainty reduction: 15.50 -> 1.25
Reduction ratio: 12.4x
```

```
Final selection distribution (last 1000):
Arm 0: 27.4%
Arm 1: 23.2%
Arm 2: 26.9%
Arm 3: 22.5%
```

1.3.4.2 Analysis The exploration dynamics reveal the classic bandit learning phases:

1. **Early exploration (0-100):** High uncertainty (3.47), uniform-ish selection, low reward (0.607)
2. **Learning (100-2000):** Uncertainty decreases (3.47 -> 1.45), reward improves (0.607 -> 0.988)
3. **Exploitation (2000-5000):** Low uncertainty (1.25), stable selection, near-optimal reward (0.992)

The **12.4x uncertainty reduction** demonstrates how Thompson Sampling naturally transitions from exploration to exploitation as it learns the reward structure.

1.3.5 Lab 6.5: Multi-Seed Robustness (5 min)

Objective: Verify bandit performance is robust across random seeds.

1.3.5.1 Solution

```
from scripts.ch06.lab_solutions import lab_6_5_multi_seed_robustness

results = lab_6_5_multi_seed_robustness()
```

```
n_seeds=5,  
n_episodes=5000,  
base_seed=42,  
verbose=True  
)
```

Actual Output:

```
=====  
Lab 6.5: Multi-Seed Robustness Analysis  
=====
```

Configuration:

```
Seeds: 5  
Episodes per seed: 5,000  
Seed 42: 0.478  
Seed 1042: 0.685  
Seed 2042: 0.446  
Seed 3042: 0.457  
Seed 4042: 0.436
```

Statistics

```
Mean: 0.500  
Std: 0.093  
CV: 18.7%  
Range: [0.436, 0.685]
```

Conclusion:

High variance (CV=18.7%) suggests sensitivity to initialization.

1.3.5.2 Analysis The multi-seed analysis reveals **higher variance than expected** (CV = 18.7%):

- **Seed 1042** achieves 0.685 (37% above mean)
- **Seed 4042** achieves 0.436 (13% below mean)

This variance comes from: 1. **Initialization effects:** Early random explorations can lead to different learning trajectories 2. **True parameter variability:** Each seed generates different true reward parameters 3. **Exploration randomness:** Stochastic action selection creates variance

Practical implication: For production deployment, either: - Run longer (more episodes to average out variance) - Use multiple seeds and report confidence intervals - Warm-start from prior knowledge to reduce initialization variance

1.4 Advanced Exercises (Optional)

1.4.1 Exercise 6.7: Hierarchical Templates (20 min)

Problem: Design a two-level bandit hierarchy: meta-bandit over objectives, sub-bandits over tactics.

1.4.1.1 Solution

```
from scripts.ch06.lab_solutions import exercise_6_7_hierarchical_templates

results = exercise_6_7_hierarchical_templates(
    n_episodes=50000,
    seed=42,
    verbose=True
)
```

Actual Output:

```
=====
Exercise 6.7: Hierarchical Templates
=====
```

Hierarchical structure:

- Level 1 (Meta): 3 objectives
 - Objective A: Maximize margin (templates: HiMargin, Premium, CM2)
 - Objective B: Maximize volume (templates: Popular, Discount, Budget)
 - Objective C: Strategic goals (templates: Strategic, Diversity)

Level 2 (Sub): 2-3 templates per objective

Training hierarchical bandit for 50,000 episodes...

Progress: 100% (50000/50000)

Meta-Level Selection Distribution:

Objective		Selection %
Margin (A)		23.6%
Volume (B)		25.2%
Strategic(C)		51.2%

Sub-Level Selection (within objectives):

Margin (A):

HiMargin		4.6%
Premium		0.5%
CM2Boost		94.9%

Volume (B):

Popular		88.5%
Discount		0.9%
Budget		10.6%

Strategic(C):

Strategic		46.6%
Diversity		53.4%

Comparison to Flat LinUCB:

Policy		Convergence (ep)
Flat LinUCB		~12,000
Hierarchical		~8,000

Convergence speedup: 33% faster (8k vs 12k episodes)

[OK] Hierarchical bandits converge faster with similar final performance.

1.4.1.2 Analysis Hierarchical templates offer two advantages:

1. **Faster convergence (33%)**: Fewer parameters per level means faster learning
2. **Better interpretability**: Business can understand “we shifted toward strategic optimization” rather than opaque template IDs

The learned hierarchy reveals interesting patterns: - **CM2Boost dominates within Margin** (94.9%): Clear winner for margin optimization - **Popular dominates within Volume** (88.5%): Popularity drives clicks/purchases - **Diversity edges out Strategic** (53.4% vs 46.6%): Close competition in strategic tier

1.4.2 Exercise 6.9: Query-Conditional Templates (30 min)

Problem: Extend templates to depend on query content, not just products.

1.4.2.1 Solution

```
from scripts.ch06.lab_solutions import exercise_6_9_query_conditional_templates

results = exercise_6_9_query_conditional_templates(
    n_episodes=30000,
    seed=42,
    verbose=True
)
```

Actual Output:

```
=====
Exercise 6.9: Query-Conditional Templates
=====
```

Query-conditional template design:

$$t(p, q) = w_{\text{base}} * f(p) + w_{\text{query}} * g(q, p)$$

where $g(q, p)$ captures query-product interaction

Training comparison (30,000 episodes):

Policy	Final Reward
Product-only templates	0.81
Query-conditional	0.98

DeltaReward: +0.16 (+20.1%)

Insight:

Query-conditional templates learn to AMPLIFY templates when query content suggests they'll be effective, and SUPPRESS templates when query content suggests they'll hurt.

This is learned automatically from reward feedback---no manual query->template rules needed.

[OK] Query-conditional templates achieve +20.1% improvement.

1.4.2.2 Analysis Query-conditional templates demonstrate the value of incorporating additional context. The **+20.1% improvement** comes from better matching templates to user intent signals in the query.

Key insight: When query says “deals,” boosting discounted products is obviously good. When query says “premium,” boosting discounts actively hurts. Query-conditional templates learn this automatically from reward signals—no manual rules needed.

1.5 Summary

These lab solutions demonstrate the core lessons of Chapter 6:

1. **Features matter more than algorithms:** The gap between simple and rich features (where it exists) dwarfs algorithm differences.
2. **Regularization matters:** The choice of regularization mode (`blend` vs `quantized`) can flip which algorithm wins.
3. **Bandits learn segment policies automatically:** Given good features, LinUCB/TS discover which templates work for which users without manual rules.
4. **Exploration has a cost:** Early regret is the price of learning. The 12.4x uncertainty reduction shows how this cost diminishes over time.

5. **Hyperparameters have sensible defaults:** $\lambda = 1.0$, $\alpha = 1.0$ work across a wide range of conditions.
 6. **Variance is real:** The 18.7% CV across seeds reminds us that bandit results should be reported with confidence intervals.
 7. **Production requires engineering:** Cholesky optimization (52.8x speedup), robust seed handling, and interpretable hierarchies matter as much as theoretical elegance.
 8. **Diversity depends on context:** Exercise 6.6 showed diversity failing (0% improvement), while Exercise 6.6b showed it succeeding (+94.4% improvement). The difference? Whether the base ranker's bias misses user preferences.
-

1.6 Running the Code

All solutions are in `scripts/ch06/lab_solutions/`:

```
# Run all exercises and labs
python -m scripts.ch06.lab_solutions --all

# Run specific exercise
python -m scripts.ch06.lab_solutions --exercise 6.1
python -m scripts.ch06.lab_solutions --exercise 6.5
python -m scripts.ch06.lab_solutions --exercise 6.6b # When diversity helps
python -m scripts.ch06.lab_solutions --exercise lab6.3

# Run production simulator experiments (requires zoosim)
python scripts/ch06/template_bandits_demo.py \
    --n-static 2000 \
    --n-bandit 20000 \
    --features simple

python scripts/ch06/template_bandits_demo.py \
    --n-static 2000 \
    --n-bandit 20000 \
    --features rich \
    --rich-regularization blend
```

Chapter 6 Lab Solutions — 2025