Mechanistic Interpretability of Code Correctness in LLMs via Sparse Autoencoders

Technical Manual

October 20, 2025

This technical manual documents the implementation for investigating how language models internally represent code correctness using Sparse Autoencoders (SAEs).

Contents

1 Introduction

1.1 Methodology Overview

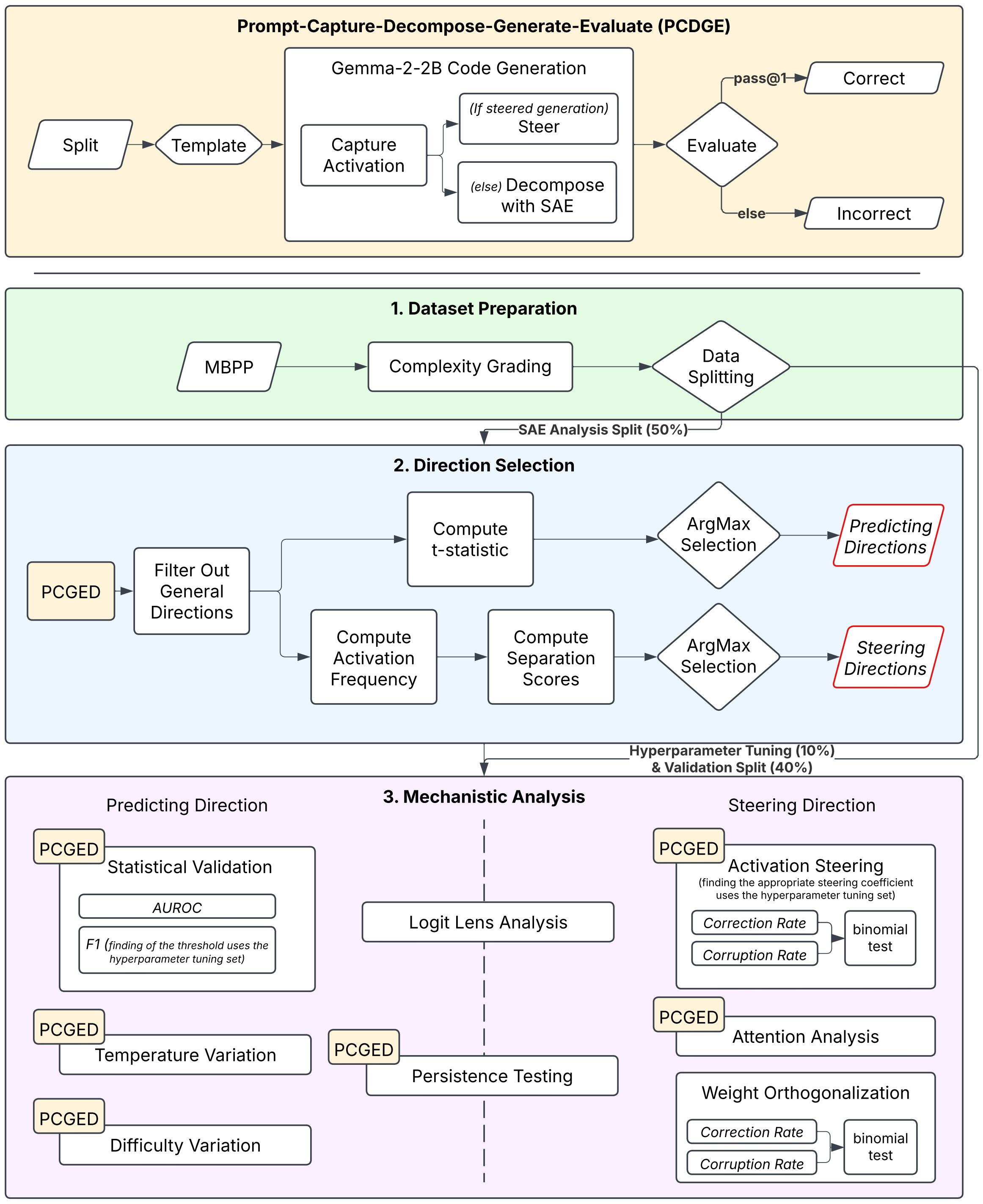

Figure 1: Complete methodology pipeline showing the three main stages: Dataset Preparation, Direction Selection, and Mechanistic Analysis.

The implementation follows a three-stage pipeline (Figure 1):

1. **Dataset Preparation**: Generate and label Python code solutions with captured activations
2. **Direction Selection**: Identify SAE features discriminating correct vs incorrect code
3. **Mechanistic Analysis**: Validate features through statistical metrics and causal interventions

1.2 Core Pattern: PCDGE

The codebase uses a consistent pattern throughout all phases:
Prompt-Capture-Decompose-Generate-Evaluate (`PCDGE`):

- **Prompt**: Build prompt from MBPP problem description and test cases
- **Capture**: Extract residual stream activations at the last prompt token
- **Decompose**: Apply SAE decomposition to transform activations into interpretable latent representations
- **Generate**: Language model generates code solution
- **Evaluate**: Execute tests to determine correctness (pass@1 metric)

This pattern is implemented in `phase1` and reused with modifications throughout the pipeline.

1.3 Complete Phase Overview

The implementation consists of 28 phases organized into the three pipeline stages shown in Figure 1. This manual focuses on core phases marked with $\star$, but all phases are listed here for completeness.

Phase	Stage	Description	Coverage
Dataset Preparation			
0	Setup	Cyclomatic complexity analysis of MBPP problems	⋆
0.1	Setup	Split dataset (50% SAE/10% hyperparam/40% validation)	⋆
1	Generation	Generate solutions with activation capture	⋆
Direction Selection			
2.2	Filtering	Cache Pile-10k activations for general language filtering	⋆
2.5	Steering	Separation score computation for steering features	⋆
2.10	Predicting	T-statistic computation for predicting features	⋆
2.15	Visualization	Layer-wise analysis plots (not in paper)	
Statistical Validation			
3.5	Generation	Temperature robustness dataset (T=0.0 to 1.4)	⋆
3.6	Generation	Hyperparameter set processing (10% split)	⋆
3.8	Metrics	AUROC and F1 evaluation on validation set	⋆
3.10	Analysis	Temperature-based AUROC/F1 analysis	⋆
3.11	Visualization	Temperature trends plot update	
3.12	Analysis	Difficulty stratification (Easy/Medium/Hard)	⋆
Steering Experiments			
4.5	Coefficient	Grid search for optimal steering coefficients	⋆
4.6	Coefficient	Golden section refinement for incorrect-steering	
4.7	Visualization	Coefficient search process plots	
4.8	Analysis	Steering effect analysis and binomial testing	⋆
4.10	Baseline	Zero-discrimination feature selection (control)	⋆
4.12	Baseline	Zero-discrimination steering experiments	⋆
4.14	Significance	Statistical significance testing	⋆
Weight Orthogonalization			
5.3	Intervention	Permanent direction removal via weight modification	⋆
5.6	Baseline	Zero-discrimination orthogonalization	
5.9	Significance	Orthogonalization statistical significance	
Attention Analysis			
6.3	Mechanistic	Attention pattern changes by prompt section	⋆
Persistence Testing (Cross-Model Transfer)			
7.3	Baseline	Instruction-tuned model baseline generation	⋆
7.6	Steering	Instruction-tuned steering effect analysis	⋆
7.9	Analysis	Universality analysis across architectures	
7.12	Metrics	Instruction-tuned AUROC/F1 evaluation	⋆

Table 1: Complete phase mapping to methodology pipeline. Phases marked with ⋆ are detailed in this manual. Visualization and refinement phases are included for completeness but not individually documented.

Note on Phase Numbering: Major phase numbers (e.g., Phase 2, Phase 4) correspond to pipeline stages. Decimal suffixes indicate specific sub-tasks within that stage (e.g., 2.5, 2.10). Some phase numbers are non-consecutive due to space are anticipated beforehand if there will be a task to be inserted in between phases or subphases.

2 Running the Code

2.1 Command Structure

All phases are executed through a unified command interface:

```
python3 run.py phase <phase_number> [options]
```

2.2 Phase Execution Examples

2.2.1 Dataset Preparation

```
# Phase 0: Analyze cyclomatic complexity of MBPP problems
python3 run.py phase 0

# Phase 0.1: Split into SAE analysis (50%), hyperparameter (10%),
#            validation (40%) sets
python3 run.py phase 0.1

# Phase 1: Generate code solutions and capture activations
# Process all 487 problems in SAE analysis split
python3 run.py phase 1

# Phase 1: Process subset for testing (first 100 problems)
python3 run.py phase 1 --start 0 --end 100
```

2.2.2 Direction Selection

```
# Phase 2.2: Cache Pile-10k activations for filtering
python3 run.py phase 2.2

# Phase 2.5: SAE analysis with separation scores
# Auto-discovers Phase 1 output
python3 run.py phase 2.5

# Phase 2.10: T-statistic based feature selection
python3 run.py phase 2.10
```

2.2.3 Mechanistic Analysis

```
# Phase 3.5: Generate solutions at different temperatures
python3 run.py phase 3.5

# Phase 3.8: Evaluate AUROC and F1 scores
python3 run.py phase 3.8

# Phase 4.5: Find optimal steering coefficients
# Run only correction experiments
python3 run.py phase 4.5 --correction-only

# Phase 4.8: Steering effect analysis on validation set
python3 run.py phase 4.8

# Phase 6.3: Attention pattern analysis
python3 run.py phase 6.3
```

2.3 Key Command-Line Options

Option	Description
`--start N`	Start processing at index N
`--end M`	End processing at index M (inclusive)
`--correction-only`	Run only correction experiments (Phase 4.5, 4.8)
`--corruption-only`	Run only corruption experiments

Table 2: Common command-line options for run.py

2.4 Checkpoint and Resume

All generation phases (`phase1`, `phase3.5`, `phase4.8`) implement automatic checkpointing:

- Saves progress every 50 problems to `checkpoint_task_N.json`
- If execution is interrupted, simply re-run the same command
- Auto-detects existing checkpoints and resumes from last completed task
- Skips already processed problems to avoid duplication

Example:

```
# Initial run (crashes at problem 250)
python3 run.py phase 1 --start 0 --end 487

# Re-run same command - automatically resumes from problem 250
python3 run.py phase 1 --start 0 --end 487
```

3 Activation Capture: Detailed Implementation

This section provides detailed explanation of the `ActivationExtractor` class, which is central to all phases that capture residual stream activations.

3.1 Complete ActivationExtractor Implementation

File: `common_simplified/activation_hooks.py`

```python
class ActivationExtractor:
    """Extract residual stream activations using PyTorch hooks."""

    def __init__(self, model: torch.nn.Module,
                 layers: List[int],
                 position: int = -1):
        """
        Args:
            model: Gemma model to extract from
            layers: Layer indices (e.g., [0,1,...,25])
            position: Token position (-1 = last prompt token)
        """
        self.model = model
        self.layers = layers
        self.position = position
        self.hooks = []
        self.activations = {}

    def setup_hooks(self) -> None:
        """Attach pre-forward hooks to specified layers."""
        self.remove_hooks()  # Clean up existing hooks

        for layer_idx in self.layers:
            # Access transformer layer
            layer = self.model.model.layers[layer_idx]

            # Register hook that captures BEFORE layer transformation
            hook = layer.register_forward_pre_hook(
                self._create_hook(layer_idx)
            )
            self.hooks.append(hook)

    def _create_hook(self, layer_idx: int) -> Callable:
        """Create hook function for a specific layer."""
        def hook_fn(module, input):
            # Only capture on FIRST forward pass (prompt processing)
            # Ignore subsequent passes during autoregressive generation
            if layer_idx not in self.activations:
                # input[0]: residual stream (batch, seq_len, hidden)
                residual_stream = input[0]

                # Extract last token: [:, -1, :] -> (1, 2048)
                activation = residual_stream[:, self.position, :]
                activation = activation.detach().clone().cpu()

                self.activations[layer_idx] = activation

            # CRITICAL: Return input unchanged for steering hooks
            return input
        return hook_fn
    def remove_hooks(self) -> None:
        """Remove all registered hooks."""
        for hook in self.hooks:
            hook.remove()
        self.hooks.clear()
        self.activations.clear()
```

3.2 Key Design Decisions

1. Pre-hook vs Post-hook:

- Uses `register_forward_pre_hook` to capture residual stream *before* layer transformation
- This gives us the input to each transformer block (what SAEs are trained on)

2. Position = -1 (Last Token):

- Extracts activation from last token of prompt (where "# Solution:" ends)
- This encodes the model's understanding before generation starts

3. First-Pass Only:

- `if layer_idx not in self.activations` ensures single capture
- We only want activations from prompt processing, not autoregressive generation

4. Return Input Unchanged:

- Returning `input` preserves modifications from previous hooks
- Critical for Phase 4.8 where steering hooks modify the residual stream

3.3 Usage Example

```
# In Phase 1 runner setup
model, tokenizer = load_model_and_tokenizer("google/gemma-2-2b")

# Create extractor for all 26 layers
extractor = ActivationExtractor(
    model=model,
    layers=list(range(26)),
    position=-1  # Last prompt token
)

# Setup hooks
extractor.setup_hooks()

# Generate - hooks capture automatically
prompt = "Write function to add\n\nassert add(1,2)==3\n\n# Solution:"
inputs = tokenizer(prompt, return_tensors="pt").to("cuda")
outputs = model.generate(**inputs, max_new_tokens=800)

# Retrieve captured activations
activations = extractor.activations
# {0: tensor([[...]]), 1: tensor([[...]]), ..., 25: tensor([[...]])}
# Each shape: (1, 2048)

extractor.remove_hooks()
```

4 Steering Hooks: Detailed Implementation

This section explains how steering interventions modify the model's residual stream.

4.1 Steering Mechanism

Steering adds a scaled SAE decoder direction to the residual stream:

$$\mathbf{x}^{\text{steered}} = \mathbf{x} + \alpha \cdot \mathbf{W}_{\text{dec}}[j, :] \tag{1}$$

Where:

- $\mathbf{x} \in \mathbb{R}^{2048}$: Original residual stream
- $\alpha \in \mathbb{R}$: Steering coefficient (controls steering strength)
- $\mathbf{W}_{\text{dec}}[j, :]$: SAE decoder direction for feature j
- $\mathbf{x}^{\text{steered}}$: Modified residual stream

4.2 create_steering_hook Implementation

File: `common/steering_metrics.py`

```python
def create_steering_hook(sae_decoder_direction: torch.Tensor,
                         coefficient: float) -> Callable:
    """
    Create PyTorch hook that adds SAE decoder direction to residual stream.

    Args:
        sae_decoder_direction: Decoder vector [d_model=2048]
        coefficient: Steering strength (positive/negative)

    Returns:
        Hook function for register_forward_pre_hook()
    """
    def hook_fn(module, input):
        # input[0]: residual stream [batch, seq_len, d_model]
        residual = input[0]

        # Create steering vector [1, 1, 2048]
        steering = sae_decoder_direction.unsqueeze(0).unsqueeze(0)
        steering = steering * coefficient

        # Add to ALL token positions via broadcasting
        residual = residual + steering.to(residual.device, residual.dtype)

        # Return modified input tuple
        return (residual,) + input[1:]

    return hook_fn
```

The function creates a closure that captures the SAE decoder direction and coefficient. When the hook executes during forward pass, it extracts the residual stream from `input[0]`, which has shape [batch, seq_len, d_model]. It reshapes the decoder direction from [2048] to [1, 1, 2048] via two `unsqueeze` operations, allowing broadcasting across all batch items and sequence positions. After scaling by the coefficient, it adds the steering vector to every token position simultaneously. The modified residual stream is returned as `input[0]` while preserving other input elements unchanged.

4.3 Usage in Phase 4.8

```
# Load model and SAE
model, tokenizer = load_model_and_tokenizer("google/gemma-2-2b")
sae = load_gemma_scope_sae(layer_idx=13, device="cuda")

# Get best feature from Phase 2.5
with open("data/phase2_5/top_20_features.json") as f:
    features = json.load(f)
best = features['correct'][0]

# Extract decoder direction
decoder_dir = sae.W_dec[best['feature']]  # Shape: [2048]

# Create and register steering hook
hook_fn = create_steering_hook(decoder_dir, coefficient=10.0)
layer = model.model.layers[13]
hook_handle = layer.register_forward_pre_hook(hook_fn)

# Generate with steering active
outputs = model.generate(**inputs, max_new_tokens=800)

# Remove hook
hook_handle.remove()
```

This code demonstrates the complete steering workflow in Phase 4.8. First, it loads the model and the SAE for the target layer. It reads the top feature identified by Phase 2.5 from the saved JSON file, then extracts the corresponding decoder direction from the SAE's weight matrix. The steering hook is created with the chosen coefficient (10.0 for correct-steering) and registered to the appropriate layer using PyTorch's hook mechanism. During generation, the hook automatically modifies the residual stream at every forward pass. After generation completes, the hook is removed to restore the model to its original state.

4.4 Measuring Steering Effects

4.4.1 Correction Rate

```
def calculate_correction_rate(results: List[Dict]) -> float:
    """Measure % of incorrect→correct transitions."""
    corrected = sum(
        1 for r in results
        if not r['baseline_passed'] and r['steered_passed']
    )
    total_incorrect = sum(
        1 for r in results
        if not r['baseline_passed']
    )
    return (corrected / total_incorrect) * 100 if total_incorrect > 0 else 0.0

# Example: 100 incorrect, 35 become correct → 35% correction rate
```

The function measures the effectiveness of ”correct” steering by counting how many initially incorrect problems become correct after intervention. It iterates through results, identifying samples where `baseline_passed` is False (initially incorrect). Among these, it counts how many have `steered_passed` as True (fixed by steering). The correction rate is the percentage: if 100 problems were initially incorrect and 35 become correct after steering, the correction rate is 35%. A rate significantly above 0% indicates the steering direction causally influences correctness.

4.4.2 Corruption Rate

```
def calculate_corruption_rate(results: List[Dict]) -> float:
    """Measure % of correct→incorrect transitions."""
    corrupted = sum(
        1 for r in results
        if r['baseline_passed'] and not r['steered_passed']
    )
    total_correct = sum(
        1 for r in results
        if r['baseline_passed']
    )
    return (corrupted / total_correct) * 100 if total_correct > 0 else 0.0

# Example: 200 correct, 10 become incorrect → 5% corruption rate
```

The function measures unwanted side effects of steering by counting how many initially correct problems become incorrect after intervention. It filters for samples where `baseline_passed` is True (initially correct), then counts how many have `steered_passed` as False (broken by steering). The corruption rate is the percentage: if 200 problems were initially correct and 10 break after steering, the corruption rate is 5%.

5 Mechanistic Analysis

This section documents validation methods for the identified directions through statistical analysis and causal interventions.

5.1 Statistical Validation

5.1.1 AUROC Computation

File: phase3_8/auroc_f1_evaluator.py

Area Under ROC Curve measures discrimination ability across all thresholds:

```
from sklearn.metrics import roc_auc_score, roc_curve

# Load feature activations on validation set
y_true = labels  # 1=correct, 0=incorrect
scores = feature_activations  # SAE feature values

# Compute AUROC
auroc = roc_auc_score(y_true, scores)

# Get ROC curve points
fpr, tpr, thresholds = roc_curve(y_true, scores)
```

AUROC > 0.7 indicates strong discrimination. Value of 0.5 is random chance, 1.0 is perfect.

5.1.2 F1 Score with Threshold Optimization

F1 score requires binary predictions, so optimal threshold must be found on hyperparameter set:

```
def find_optimal_threshold(y_true, scores):
    """Find threshold maximizing F1 on hyperparameter set."""
    thresholds = np.linspace(scores.min(), scores.max(), 100)
    f1_scores = []

    for threshold in thresholds:
        y_pred = (scores >= threshold).astype(int)
        f1 = f1_score(y_true, y_pred, zero_division=0)
        f1_scores.append(f1)

    # Select threshold with max F1
    optimal_idx = np.argmax(f1_scores)
    optimal_threshold = thresholds[optimal_idx]

    return optimal_threshold

# Find on hyperparameter set
optimal_threshold = find_optimal_threshold(
    y_true_hyperparam, scores_hyperparam
)

# Evaluate on validation set
y_pred_val = (scores_val >= optimal_threshold).astype(int)
f1_val = f1_score(y_true_val, y_pred_val)
```

```
precision_val = precision_score(y_true_val, y_pred_val)
recall_val = recall_score(y_true_val, y_pred_val)
```

The function searches for the best threshold to convert continuous activation scores into binary predictions. It tests 100 evenly-spaced thresholds between the minimum and maximum activation values. For each threshold, it classifies scores $\geq$ threshold as "correct" and computes the F1 score. It returns the threshold that achieved the highest F1. This threshold is then used to make predictions on the validation set.

5.1.3 Example Output

```
Correct-Predicting Feature (Layer 16, Feature 14439):
  Hyperparameter Optimal Threshold: 10.8109
  Validation AUROC: 0.6430
  Validation F1: 0.5039
  Validation Precision: 0.3623
  Validation Recall: 0.8276
```

5.2 Temperature Variation Analysis

File: `phase3_5_temperature_robustness/temperature_runner.py`

Tests if predicting directions maintain statistical significance across different sampling temperatures.

5.2.1 Methodology

Activations captured once at temperature 0 (deterministic). Then generate multiple solutions at different temperatures:

```
# Temperature 0: capture activations (single generation)
extractor.setup_hooks()
outputs = model.generate(..., temperature=0.0, do_sample=False)
activations = extractor.get_activations()
extractor.remove_hooks()

# Temperatures 0.2, 0.4, ..., 1.4: sample without re-capturing
for temp in [0.2, 0.4, 0.6, 0.8, 1.0, 1.2, 1.4]:
    for sample in range(3):  # 3 generations per temperature
        outputs = model.generate(..., temperature=temp, do_sample=True)
        # Evaluate each generation
```

The code implements a two-phase evaluation strategy. In the first phase, activations are captured once at temperature 0 (deterministic, greedy decoding) to ensure consistent feature values across all temperature conditions. In the second phase, the model generates solutions at 7 different temperatures (0.2 to 1.4) with sampling enabled, producing 3 solutions per temperature for a total of 21 generations per problem. The activations from temperature 0 are reused to predict correctness for all temperature-varied generations, testing whether the features remain predictive under increased stochasticity.

5.2.2 Analysis

Compute AUROC and F1 for each temperature to see if predictions remain valid:

```
Temperature  AUROC   F1
0.0          0.643   0.504
0.2          0.629   0.485
0.4          0.641   0.446
0.6          0.638   0.390
0.8          0.655   0.343
1.0          0.671   0.193
1.2          0.683   0.066
1.4          0.706   0.025
```

Decreasing performance at higher temperatures is expected due to increased stochasticity, but directions should remain predictive.

5.3 Difficulty Variation Analysis

Tests if predicting directions generalize across problem complexity levels.

5.3.1 Complexity Stratification

Problems categorized by McCabe's cyclomatic complexity:

```python
# Stratify by complexity
easy = df[df['cyclomatic_complexity'] == 1]
medium = df[df['cyclomatic_complexity'].between(2, 3)]
hard = df[df['cyclomatic_complexity'] >= 4]

# Compute metrics for each group
for complexity_group in [easy, medium, hard]:
    auroc = roc_auc_score(complexity_group['labels'],
                          complexity_group['activations'])
    f1 = f1_score(...)
```

The code partitions the validation set into three difficulty tiers based on cyclomatic complexity scores computed in Phase 0. Easy problems have complexity exactly 1 (simple, linear code), medium problems have complexity 2-3 (moderate branching), and hard problems have complexity 4 or higher (complex control flow). For each tier, the code computes AUROC and F1 scores independently, allowing analysis of whether the predicting directions generalize across problem difficulty levels or only work for specific complexity ranges.

5.3.2 Example Results

```
Complexity     n_samples   AUROC   F1
Easy (=1)      106         0.641   0.638
Medium (2-3)   181         0.586   0.456
Hard (>=4)     101         0.679   0.361
```

5.4 Activation Steering with Coefficient Search

5.4.1 Grid Search for Optimal Coefficient

File: `phase4_5_model_steering/steering_coefficient_selector.py`

Searches for coefficient maximizing correction/corruption rate on hyperparameter set:

```python
def simple_grid_search(steering_type):
    """Test coefficients from 10 to 100 in steps of 10."""
    grid_points = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
    best_coefficient = 10
    best_score = 0

    for coeff in grid_points:
        # Create steering hook
        hook_fn = create_steering_hook(decoder_direction, coeff)
        layer.register_forward_pre_hook(hook_fn)

        # Test on hyperparameter set
        if steering_type == 'correct':
            score = calculate_correction_rate(results)
        else:
            score = calculate_composite_score(results)

        if score > best_score:
            best_score = score
            best_coefficient = coeff

        # Early stopping if performance drops
        if found_peak and score < best_score:
            break

    return best_coefficient
```

The function tests coefficients 10, 20, 30... up to 100. For each coefficient, it creates a steering hook, runs the model on all problems in the hyperparameter set, and calculates the correction rate (for correct steering) or composite score (for incorrect steering). It tracks the best coefficient seen so far. If the score drops after finding a peak, it stops early to save time. Returns the coefficient with the highest score.

5.5 Binomial Testing

5.5.1 Three-Setup Comparison

Statistical validation uses three conditions:

```python
# Baseline: No intervention (0% correction/corruption expected)
baseline_results = generate_without_steering()

# Control: Random feature from different layer (noise check)
random_feature_idx = random.choice(zero_separation_features)
control_results = generate_with_steering(random_feature_idx, coeff)

# Steering: Our identified feature
steering_results = generate_with_steering(best_feature_idx, coeff)
```

The experimental design uses three conditions to isolate the causal effect of the identified feature. The baseline condition generates code without any intervention, establishing the natural correction/corruption rate (typically ∼0%). The control condition steers using a randomly selected zero-discrimination feature (one with zero separation score), checking whether any random direction could produce similar effects. The steering condition uses the identified high-separation feature.

Comparing steering vs. baseline tests for effect existence, while comparing steering vs. control tests for effect specificity.

5.5.2 Statistical Test

```python
from scipy.stats import binomtest

# Test steering vs baseline
n_trials = len(initially_incorrect)
n_successes = sum(1 for r in steering_results if r['flipped_correct'])
baseline_p = 0.0  # Baseline correction rate

result = binomtest(n_successes, n_trials, baseline_p, alternative='greater')

if result.pvalue < 0.05:
    print(f"Significant: {n_successes}/{n_trials} corrected, p={result.pvalue:.4f}")
```

The binomial test checks if the observed correction rate is statistically significant. `n_trials` is the total number of initially incorrect problems. `n_successes` is how many flipped to correct after steering. `baseline_p` is the expected rate under the null hypothesis (0% for baseline). The test returns a p-value: if $p < 0.05$, the steering effect is statistically significant and not due to chance.

5.5.3 Example Output

```
Correct Steering (coefficient=29):
  Correction Rate: 11/272 (4.0%)
  p-value: <0.001 (significant)
```

5.6 Attention Analysis

File: `phase6_3_attention_analysis/attention_analyzer.py`

Analyzes how steering redistributes attention across prompt sections.

5.6.1 Extraction

Extract attention weights from all heads at steering layer:

```python
class AttentionExtractor:
    def _create_hook(self, layer_idx):
        def hook_fn(module, input, output):
            # output.attentions: [batch, n_heads, seq_len, seq_len]
            # Extract attention from last prompt token
            attn = output.attentions[:, :, -1, :]  # [batch, n_heads, seq_len]
            self.attention_patterns[layer_idx] = attn.detach().cpu()
        return hook_fn
```

The hook function captures attention weights from the model's output. `output.attentions` has shape [batch, n_heads, seq_len, seq_len] representing attention from each token to every other token. We slice [:, :, -1, :] to get attention FROM the last prompt token TO all tokens, giving [batch, n_heads, seq_len]. This shows where the model is "looking" at the final prompt position.

5.6.2 Aggregation by Section

Sum attention within each prompt section:

```python
def aggregate_to_sections(attention_tensor, boundaries):
    """
    Args:
        attention_tensor: [n_heads, seq_len]
        boundaries: {'problem_end': int, 'test_end': int}

    Returns:
        Attention percentages for each section
    """
    section_attn = {
        'problem': attention_tensor[:, :boundaries['problem_end']].sum(-1),
        'tests': attention_tensor[:,
        ↪  boundaries['problem_end']:boundaries['test_end']].sum(-1),
        'solution': attention_tensor[:, boundaries['test_end']:].sum(-1)
    }

    # Normalize to percentages
    total = sum(section_attn.values())
    return {k: (v / total * 100) for k, v in section_attn.items()}
```

The function takes raw attention weights for all sequence positions and groups them into three sections based on boundaries. It sums attention weights within each section (e.g., all tokens from 0 to `problem_end` go into 'problem'). Then it normalizes by the total attention to get percentages. Result: each section's percentage represents how much the model attends to that part of the prompt.

5.6.3 Computing Changes

```python
# Average across tasks
baseline_attn = aggregate_all_tasks(baseline_attention_data)
steered_attn = aggregate_all_tasks(steered_attention_data)

# Percentage point changes
delta_problem = steered_attn['problem'] - baseline_attn['problem']
delta_tests = steered_attn['tests'] - baseline_attn['tests']
delta_solution = steered_attn['solution'] - baseline_attn['solution']
```

The code computes percentage point changes in attention allocation between baseline and steered conditions. First, it aggregates attention patterns across all validation tasks for each condition, averaging the section-wise percentages to get mean baseline and steered attention distributions. Then it calculates simple differences: if baseline allocates 30% attention to tests and steered allocates 44.6% attention to tests, the change is +14.6 percentage points. These delta values quantify how steering mechanistically shifts the model's focus across different prompt sections.

5.6.4 Example Results

```
Correct Steering Attention Changes:
  Problem Description: -7.75% (p<0.001, significant decrease)
  Test Cases: +14.60% (p<0.001, significant increase)
  Solution Marker: -6.85% (p<0.001, significant decrease)
```

5.7 Weight Orthogonalization

File: `phase5_3_weight_orthogonalization/weight_orthogonalizer.py`

Permanently removes a direction from all matrices writing to residual stream.

5.7.1 Orthogonalization Formula

```
def orthogonalize_weights(model, direction, layer_idx):
    """
    Modify W_out to prevent writing direction d.

    W_new = W_old - W_old @ d.T @ d
    """
    layer = model.model.layers[layer_idx]

    # Normalize direction
    d = direction / torch.norm(direction)

    # Orthogonalize all output matrices
    for module in [layer.self_attn.o_proj, layer.mlp.down_proj]:
        W_out = module.weight.data
        # Project out direction
        W_out -= torch.outer(W_out @ d, d)

    return model
```

The function modifies weight matrices in-place to remove a direction. First it normalizes the direction vector to unit length. Then for each output matrix (attention output and MLP output), it subtracts the component that writes in the direction `d`. The formula `W -= outer(W @ d, d)` projects `d` out of `W`'s column space. After this, the layer cannot write the direction to the residual stream.

5.7.2 Testing Setup

Same three-condition framework as steering:

```
# Baseline: Unmodified model
baseline_model = load_model()

# Control: Orthogonalize random zero-separation feature
control_model = orthogonalize_weights(load_model(), random_direction, layer)

# Orthogonalization: Orthogonalize our identified feature
ortho_model = orthogonalize_weights(load_model(), target_direction, layer)
```

The code mirrors the three-condition design used for steering experiments. The baseline uses an unmodified model to establish natural performance. The control orthogonalizes a random zero-discrimination feature to verify that arbitrary direction removal doesn't cause similar degradation. The orthogonalization condition removes the identified correctness direction. This design allows causal testing: if orthogonalizing the target direction significantly degrades performance while the control does not, it proves the direction is causally necessary for code generation.

5.7.3 Example Results

```
Correct Direction Orthogonalization:
  Corruption of initially correct: 97/116 (83.6%)
  p-value: <0.001 (significant)

Interpretation: Removing correct direction severely degrades
code quality, proving it's causally necessary for generation.
```

5.8 Persistence Testing

Tests whether directions identified in base model transfer to instruction-tuned variant.

5.8.1 Methodology

```python
# 1. Identify feature in base model (google/gemma-2-2b)
base_model = "google/gemma-2-2b"
best_feature = phase2_10_results['correct'][0]  # Layer 13, Feature 8472

# 2. Apply SAME feature to instruction-tuned model
instruct_model = "google/gemma-2-2b-it"
# Use same layer (13), same feature_idx (8472), same coefficient (10.0)

# 3. Evaluate with identical metrics
f1_base = evaluate_predicting_direction(base_model, best_feature)
f1_instruct = evaluate_predicting_direction(instruct_model, best_feature)

correction_rate_base = evaluate_steering(base_model, best_feature, coeff=10.0)
correction_rate_instruct = evaluate_steering(instruct_model, best_feature, coeff=10.0)
```

The code identifies the best feature on the base model (e.g., google/gemma-2-2b). Then it loads the instruction-tuned version (google/gemma-2-2b-it) and applies the SAME feature (same layer index, same feature index, same coefficient). It evaluates both models using identical metrics (F1 for predicting, correction rate for steering). Comparing the scores shows how well the feature transfers across fine-tuning.

5.8.2 Example Results

```
Predicting Direction Transfer (Layer 16, Feature 14439):
  Base Model (gemma-2-2b):
    AUROC: 0.6430, F1: 0.5039
  Instruction-Tuned (gemma-2-2b-it):
    AUROC: 0.5555, F1: 0.3817
  Retention: 86.4% (AUROC), 75.7% (F1)

Steering Direction Transfer (Layer 16, Feature 11225):
  Base Model Correction Rate: 4.0%
  Instruction-Tuned Correction Rate: 2.9%
  Retention: 72.4%

Conclusion: Directions show partial transfer to instruction-tuned
model with moderate degradation, suggesting representations are
partially restructured during fine-tuning.
```

6 Direction Selection

This section documents how code correctness directions are identified from SAE features using statistical metrics.

6.1 T-Statistic Computation

File: phase2_10_t_statistic_latent_selector/t_statistic_selector.py

T-statistics identify features that best discriminate between correct and incorrect code by measuring effect size normalized by variance. Uses Welch's t-test which handles unequal variances.

```python
def compute_t_statistics(
    correct_features: torch.Tensor,
    incorrect_features: torch.Tensor
) -> Dict[str, List[float]]:
    """
    Calculate t-statistics for predicting directions.

    Args:
        correct_features: [n_correct_samples, 16384]
        incorrect_features: [n_incorrect_samples, 16384]

    Returns:
        Dict with t_stats_correct and t_stats_incorrect lists
    """
    t_stats_correct = []
    t_stats_incorrect = []

    n_features = correct_features.shape[1]

    for i in range(n_features):
        correct_acts = correct_features[:, i].cpu().numpy()
        incorrect_acts = incorrect_features[:, i].cpu().numpy()

        # Welch's t-test (unequal variances)
        t_stat_correct = stats.ttest_ind(
            correct_acts,
            incorrect_acts,
            equal_var=False
        ).statistic

        # Swap order for incorrect direction
        t_stat_incorrect = stats.ttest_ind(
            incorrect_acts,
            correct_acts,
            equal_var=False
        ).statistic

        t_stats_correct.append(float(t_stat_correct))
        t_stats_incorrect.append(float(t_stat_incorrect))

    return {
        't_stats_correct': t_stats_correct,
        't_stats_incorrect': t_stats_incorrect
    }
```

The function loops through all 16,384 SAE features. For each feature, it extracts activation values from correct and incorrect samples, then runs Welch's t-test comparing the two groups. It computes two scores: one testing if correct activations are higher (`t_stats_correct`), and one testing if incorrect activations are higher (`t_stats_incorrect`). Returns both lists of t-statistics. Higher t-statistics indicate stronger discrimination. Positive t-stat for correct direction means feature activates more for correct code.

6.2 Separation Score Computation

File: `phase2_5_simplified/sae_analyzer.py`

Separation scores measure categorical exclusivity for steering directions. High separation indicates switch-like features.

```python
def compute_separation_scores(
    correct_features: torch.Tensor,
    incorrect_features: torch.Tensor
) -> Dict[str, np.ndarray]:
    """
    Calculate separation scores for steering directions.

    Returns:
        Dictionary with activation frequencies and separation scores
    """
    # Compute activation frequencies
    n_correct = correct_features.shape[0]
    n_incorrect = incorrect_features.shape[0]

    freq_correct = (correct_features > 0).float().mean(dim=0).cpu().numpy()
    freq_incorrect = (incorrect_features > 0).float().mean(dim=0).cpu().numpy()

    # Separation = difference in activation frequencies
    separation_correct = freq_correct - freq_incorrect
    separation_incorrect = freq_incorrect - freq_correct

    return {
        'separation_correct': separation_correct,
        'separation_incorrect': separation_incorrect,
        'freq_correct': freq_correct,
        'freq_incorrect': freq_incorrect
    }
```

The function counts how often each feature activates (value > 0) in correct samples versus incorrect samples. `freq_correct` is the percentage of correct samples where the feature fires. `freq_incorrect` is the same for incorrect samples. Separation score is simply the difference: if a feature fires 80% for correct but only 20% for incorrect, separation $= 0.60$. High separation indicates switch-like features good for steering.

6.3 ArgMax Selection

After computing metrics across all 26 layers, select features with highest scores:

```
# For predicting directions (Phase 2.10)
correct_predicting = argmax(t_stats_correct across all layers and features)
```

```
incorrect_predicting = argmax(t_stats_incorrect across all layers and features)

# For steering directions (Phase 2.5)
correct_steering = argmax(separation_correct across all layers and features)
incorrect_steering = argmax(separation_incorrect across all layers and features)
```

After running t-statistic computation across all 26 layers ($26 \times 16{,}384 = 425{,}984$ total features), we select the single feature with the highest t-statistic for predicting. Similarly, after computing separation scores across all layers, we select the feature with highest separation for steering. This gives us 4 features total: correct-predicting, incorrect-predicting, correct-steering, incorrect-steering.

6.4 Pile Filtering

Features activating >2% on Pile-10k dataset are filtered out to remove general language patterns:

```
# Phase 2.2 caches Pile activations
pile_freq = (pile_features > 0).float().mean(dim=0)

# Filter features
code_specific_features = features[pile_freq < 0.02]
```

The code implements a simple threshold-based filter to remove general language features. It computes the activation frequency for each SAE feature on the Pile-10k dataset (what percentage of Pile samples activate each feature). Features that fire on more than 2% of general text samples are discarded, as they likely encode generic linguistic patterns rather than code-specific concepts. Only features with `pile_freq` < 0.02 are retained, ensuring the selected directions are specialized for code correctness rather than general language understanding.

6.5 Example Output

Phase 2.10 selects top 20 features per direction and saves to `top_20_features.json`:

```
{
  "correct": [
    {"layer": 16, "feature_idx": 14439, "t_statistic": 5.09},
    {"layer": 16, "feature_idx": 11225, "t_statistic": 5.06},
    {"layer": 10, "feature_idx": 513, "t_statistic": 4.91}
  ],
  "incorrect": [
    {"layer": 19, "feature_idx": 5441, "t_statistic": 5.68},
    {"layer": 22, "feature_idx": 13837, "t_statistic": 4.77},
    {"layer": 20, "feature_idx": 9996, "t_statistic": 4.64}
  ]
}
```

7 SAE Decomposition Details

7.1 JumpReLU SAE Implementation

File: phase2_5_simplified/sae_analyzer.py

```python
class JumpReLUSAE(torch.nn.Module):
    """JumpReLU Sparse Autoencoder from GemmaScope."""

    def __init__(self, d_model: int, d_sae: int):
        """
        Args:
            d_model: Hidden dimension (2048 for Gemma-2-2B)
            d_sae: SAE feature dimension (16384 for 16k SAE)
        """
        super().__init__()
        self.d_model = d_model
        self.d_sae = d_sae

        # Encoder projects residual to SAE features
        self.W_enc = torch.nn.Parameter(torch.zeros(d_model, d_sae))
        self.b_enc = torch.nn.Parameter(torch.zeros(d_sae))

        # Decoder projects SAE features back
        self.W_dec = torch.nn.Parameter(torch.zeros(d_sae, d_model))
        self.b_dec = torch.nn.Parameter(torch.zeros(d_model))

        # JumpReLU threshold (learned per feature)
        self.threshold = torch.nn.Parameter(torch.zeros(d_sae))

    def encode(self, x: torch.Tensor) -> torch.Tensor:
        """Encode to sparse features.

        Args:
            x: Residual stream [batch, d_model]

        Returns:
            Sparse features [batch, d_sae]
        """
        # Pre-activation
        pre_acts = x @ self.W_enc + self.b_enc

        # JumpReLU: threshold then ReLU
        mask = (pre_acts > self.threshold)
        acts = mask * torch.nn.functional.relu(pre_acts)

        return acts
```

The encode method implements a two-step sparse activation mechanism. First, it computes pre-activations by projecting the input through the encoder weights and adding the bias term. Second, it applies the JumpReLU nonlinearity: for each feature, it checks if the pre-activation exceeds the learned threshold. If yes, it applies ReLU (setting negative values to zero but keeping positive values); if no, it sets the feature to zero regardless of the pre-activation value. This creates a sparse representation where most features are exactly zero, and only features with strong evidence (above their learned threshold) activate.

7.2 Computing Separation Scores

```python
def compute_separation_scores(
    correct_features: torch.Tensor,
    incorrect_features: torch.Tensor
) -> Dict[str, np.ndarray]:
    """
    Calculate separation between correct and incorrect distributions.

    Args:
        correct_features: [n_correct, 16384]
        incorrect_features: [n_incorrect, 16384]

    Returns:
        Dictionary with separation scores
    """
    # Mean activation per feature
    mean_correct = correct_features.mean(dim=0).cpu().numpy()
    mean_incorrect = incorrect_features.mean(dim=0).cpu().numpy()

    # Separation scores
    # Positive = feature activates more for correct
    separation_correct = mean_correct - mean_incorrect
    separation_incorrect = mean_incorrect - mean_correct

    return {
        'separation_correct': separation_correct,
        'separation_incorrect': separation_incorrect,
        'mean_correct': mean_correct,
        'mean_incorrect': mean_incorrect
    }
```

This implementation uses mean activation values rather than activation frequencies. For each feature, it computes the average activation magnitude across all correct samples and all incorrect samples. The separation score is the difference between these means: if a feature has mean activation 0.8 for correct code but 0.2 for incorrect code, the separation is 0.6. This approach captures both the frequency and magnitude of activations.

8 Conclusion

This technical manual documents the complete implementation pipeline for investigating code correctness representations using Sparse Autoencoders.

8.1 Key Components

1. **Activation Capture**: `ActivationExtractor` uses PyTorch pre-forward hooks to capture residual stream at last prompt token
2. **SAE Decomposition**: GemmaScope JumpReLU SAEs (16k features) decompose activations into sparse features
3. **Feature Selection**: Separation scores, T-statistic and Pile filtering identify code-specific correctness features

4. **Steering Intervention**: `create_steering_hook` adds scaled SAE decoder directions to residual stream
5. **Statistical Validation**: AUROC/F1 metrics and binomial tests validate discrimination and steering effects