Mathematical Error Fingerprinting: A Novel Method for LLM Identification

Abstract

This research began with a simple observation: Large Language Models (LLMs) consistently fail to perform accurate mathematical calculations despite their theoretical computational capabilities. While investigating this phenomenon, I discovered that different models exhibit distinctive error patterns when performing identical calculations which essentially created a unique "fingerprint" that can identify specific models. Building on this insight, I developed a framework for model identification based solely on mathematical error patterns, allowing users to test which specific model might be used in a service. This paper traces my journey from initial curiosity about mathematical errors to the development and validation of a practical identification system, providing both a new tool and insights into how mathematical knowledge is represented within these models.

1. Introduction

1.1 The Mathematical Paradox of LLMs

My research journey began with a puzzling observation: despite the impressive capabilities of LLMs across various domains, they consistently fail at basic arithmetic operations. When testing a local deployment of Llama 3, I observed that while it could perform reasonably well on 2-3 digit multiplication problems, its error rate increased dramatically with digit length. This observation seemed paradoxical given that these models excel at complex reasoning tasks, including advanced mathematical problems that require conceptual understanding.

1.2 Initial Investigation

My Project proposal experiments documented this phenomenon, revealing a clear pattern:

1. For calculations with 1-4 digits, error rates remained relatively low
2. For 5+ digit numbers, error rates increased catastrophically

Relative Error Statistics by Category:

Category	mean	median	std	min	max
1_digit	1.400000e+01	0.00	3.505000e+01	0.00	1.000000e+02
2_digit	2.016480e+03	0.47	1.408347e+04	0.04	9.960947e+04
3_digit	2.011407e+09	0.44	1.422242e+10	0.00	1.005677e+11
4_digit	2.020000e+01	0.28	4.029000e+01	0.01	1.000000e+02
5_digit	2.248650e+03	89.99	1.412532e+04	0.01	9.976608e+04
6_digit	4.460170e+03	99.90	1.979468e+04	0.01	1.005276e+05
7_digit	1.990619e+09	99.90	1.407580e+10	0.02	9.953091e+10
8_digit	1.719600e+02	100.00	2.460300e+02	0.09	9.025800e+02
9_digit	2.221564e+07	99.90	1.412477e+08	0.12	9.952236e+08

Figure 1 from my preliminary work illustrates this trend: The overall pattern suggests that as the number of digits increases, the relative errors become larger and more variable, with a particularly significant jump in both central tendency and spread around the 4-5 digit mark.

1.3 The Key Insight: Mathematical Error as a Fingerprint

While investigating potential solutions to this problem, I made a discovery: the error patterns were not random but highly consistent within a given model and distinctly different between models. I realized that these unique error patterns could serve as a "fingerprint" to identify specific models. This insight shifted my research focus from merely understanding mathematical limitations to developing an identification methodology.

1.4 Research Objectives

My research evolved to address the following specific objectives:

1. Create a reliable mathematical fingerprint that can identify specific LLMs
2. Determine the minimum computational resources (number of tests/API calls) needed to accurately identify a model
3. Establish a practical methodology for model identification that could be deployed with minimal overhead

2. Related Work

2.1 Neural Network Computational Theory

My work builds on fundamental research establishing the theoretical computational capabilities of neural networks. Siegelmann & Sontag (1992) proved that neural networks with matrix operations and simple non-linear activation functions are Turing complete, capable of universal

computation. Bukatin & Anthony (2017) extended this work to show that more general matrix-based systems like dataflow matrix machines achieve similar computational power. These findings provide the theoretical foundation for my investigation: if neural networks are theoretically capable of universal computation, why do they struggle with basic arithmetic?

2.2 LLM Mathematical Capabilities

Despite their impressive performance on many mathematical reasoning tasks, LLMs show consistent limitations in arithmetic calculations. This paradox has been documented in evaluation benchmarks like GPQA, where models demonstrate PhD-level mathematical reasoning yet struggle with basic computations. Palamas (2017) specifically investigated neural networks' ability to learn modular arithmetic, finding significant limitations. My work extends this understanding by systematically characterizing these limitations across different models and leveraging these patterns for model identification.

2.3 Superposition in Neural Networks

The concept of superposition (Elhage et al., 2022) offers a theoretical framework for understanding my observations. This theory suggests that neural networks represent more features than they have dimensions by encoding multiple concepts in overlapping patterns. This creates interference that manifests as systematic errors in certain contexts. My research on mathematical error patterns provides evidence supporting this theory, showing how the distinctive ways different models handle this superposition create identifiable fingerprints.

3. Methodology

3.1 Creating the Fingerprint

My approach to model identification relies on capturing distinctive mathematical error patterns that serve as fingerprints for different LLMs. The methodology involves a four-phase process: test generation, data collection, fingerprint construction, and similarity-based identification.

3.1.1 Test Generation

I created a diverse set of mathematical problems designed to reveal distinctive error patterns:

```
# From test_generator.py
def generate_comprehensive_tests(self, num_tests_per_category: int = 5) -> List[Dict[str, Any]]:
    """Generate a comprehensive test suite covering various mathematical patterns."""
    test_cases = []

    # 1. Standard tests with varying digit lengths
    for digits in range(2, 9):
        test_cases.extend(self._generate_standard_tests(
```

```
            num_tests_per_category,
            min_digits=digits,
            max_digits=digits
        ))

    # 2. Powers of 10
    test_cases.extend(self._generate_power_of_10_tests(num_tests_per_category))

    # 3. Repeated digits
    test_cases.extend(self._generate_repeated_digit_tests(num_tests_per_category))

    # 4. Near powers of 10
    test_cases.extend(self._generate_near_power_of_10_tests(num_tests_per_category))

    # 5. Repeating patterns
    test_cases.extend(self._generate_repeating_pattern_tests(num_tests_per_category))

    # 6. Edge cases
    test_cases.extend(self._generate_edge_cases(num_tests_per_category))

    # Shuffle tests to avoid any bias from order
    self.rng.shuffle(test_cases)
    return test_cases
```

This approach ensures coverage of:

- Basic arithmetic operations with varying digit lengths (2-8 digits)
- Pattern-based tests with powers of 10, repeated digits, and near-powers of 10
- Edge cases designed to stress computational boundaries

3.1.2 Data Collection

For each test case, I query the model and collect detailed error metrics:

```
# From api_client.py
def run_test(self, model: str, test_case: Dict[str, Any], use_cache: bool = True) -> Dict[str, Any]:
    """Run a specific test case on a model."""
    num1 = test_case["num1"]
    num2 = test_case["num2"]

    # Construct prompt
    prompt = self._generate_test_prompt(num1, num2)

    # Query the model
```

```
    api_response = self.query_model(model, prompt, use_cache)

    # Extract number from response
    predicted_result = self._extract_number(response_text)

    # Calculate error metrics
    true_result = test_case["true_result"]
    absolute_error = abs(predicted_result - true_result)
    relative_error = (absolute_error / true_result) * 100 if true_result != 0 else float('inf')

    # Analyze digit-by-digit error pattern
    digit_errors = self._analyze_digit_errors(predicted_result, true_result)

    return {
        "model": model,
        "num1": num1,
        "num2": num2,
        "true_result": true_result,
        "predicted_result": predicted_result,
        "absolute_error": absolute_error,
        "relative_error": relative_error,
        "digits": test_case["digits"],
        "test_type": test_type,
        "digit_errors": digit_errors,
        "digit_error_count": sum(1 for e in digit_errors if e != 0),
        "raw_response": response_text,
        "cached": api_response.get("cached", False)
    }
```

This captures detailed data including:

- Absolute and relative errors
- Digit-by-digit error analysis
- Error metrics categorized by test type and digit length

3.1.3 Fingerprint Construction

From these test results, I extract distinctive features that form a unique model fingerprint:

```
# From fingerprint_module.py
def generate_from_results(self, results: List[Dict[str, Any]]) -> Dict:
    """Generate a fingerprint from a list of test results."""
    # Convert results to DataFrame for analysis
    df = pd.DataFrame(results)
```

```
    valid_data = df.dropna(subset=['relative_error'])

    # 1. Error rates by digit length
    error_by_digit = valid_data.groupby('digits')['relative_error'].mean().to_dict()
    self.fingerprint['error_by_digit'] = error_by_digit

    # 2. Error growth rates between digit lengths
    growth_rates = {}
    for i in range(3, 9):
        if i-1 in error_by_digit and i in error_by_digit:
            prev_error = error_by_digit[i-1]
            curr_error = error_by_digit[i]
            if prev_error > 0:
                growth_rates[i] = curr_error / prev_error

    self.fingerprint['growth_rates'] = growth_rates
    self.fingerprint['error_growth_rate'] = np.mean(list(growth_rates.values())) if growth_rates else
None

    # 3. Direction bias (tendency to over/underestimate)
    true_vs_predicted = valid_data[['true_result', 'predicted_result']].dropna()
    if not true_vs_predicted.empty:
        true_vs_predicted['error_direction'] = true_vs_predicted.apply(
            lambda row: 1 if row['predicted_result'] > row['true_result'] else
                    (-1 if row['predicted_result'] < row['true_result'] else 0),
            axis=1
        )
        self.fingerprint['direction_bias'] = true_vs_predicted['error_direction'].mean()

    # 4. Performance on different test types
    type_performance = {}
    for test_type in valid_data['test_type'].unique():
        type_data = valid_data[valid_data['test_type'] == test_type]
        if not type_data.empty:
            type_performance[test_type] = {
                'accuracy': (type_data['absolute_error'] == 0).mean(),
                'avg_error': type_data['relative_error'].mean()
            }

    self.fingerprint['test_type_performance'] = type_performance

    # 5. Overall accuracy rate
    self.fingerprint['accuracy_rate'] = (valid_data['absolute_error'] == 0).mean()
```

```
    # 6. Digit-level error patterns
    self._analyze_digit_error_patterns(valid_data)

    return self.fingerprint
```

The key components of the fingerprint include:

- Error distribution by digit length
- Error growth rate between digit thresholds
- Direction bias (whether a model tends to overestimate or underestimate)
- Performance on different test types
- Digit-level error patterns

3.2 Test Battery Design and Efficiency Optimization

To maximize identification accuracy while minimizing API calls, I developed a hierarchical testing approach:

```
# From model_identifier.py
def identify_model(self, unknown_model: str, verbose: bool = True) -> Dict[str, Any]:
    """Identify an unknown model by comparing its fingerprint to known models."""
    # Stage 1: Initial screening with a small test set
    screening_tests = self.test_generator.generate_screening_tests(num_tests=10)
    screening_results = self.api_client.run_tests(unknown_model, screening_tests)

    # Generate preliminary fingerprint
    unknown_fp = ModelFingerprint(unknown_model)
    unknown_fp.generate_from_results(screening_results)

    # Compare with known fingerprints
    comparison = FingerprintComparison(self.fp_database.fingerprints)
    initial_matches = comparison.compare(unknown_fp)

    # Check if we have a confident match already
    top_match, top_score, _ = initial_matches[0]

    if top_score >= self.confidence_threshold:
        return {
            "identified_model": top_match,
            "confidence": top_score,
            "tests_run": len(screening_tests),
            "api_calls": self.total_api_calls,
            "all_matches": initial_matches
        }
```

```
    # Stage 2: Targeted testing for similar models
    competing_models = []
    for model, score, _ in initial_matches:
        if score > top_score - 0.2:  # Within 0.2 of top score
            competing_models.append(model)

    # Generate targeted tests for competing models
    targeted_tests = []
    for i in range(len(competing_fps) - 1):
        model1 = competing_models[i]
        for j in range(i+1, len(competing_fps)):
            model2 = competing_models[j]
            fp1 = competing_fps[model1].fingerprint
            fp2 = competing_fps[model2].fingerprint

            # Generate tests that differentiate these two models
            model_tests = self.test_generator.generate_targeted_tests(fp1, fp2, num_tests=5)
            targeted_tests.extend(model_tests)
```

The system uses a two-stage approach:

1. **Initial Screening**: Start with just 10 carefully selected tests
2. **Targeted Testing**: Only if necessary, run additional tests specifically designed to differentiate between similar models

To further minimize API calls, I implemented adaptive test selection:

```
# From test_generator.py
def _identify_differentiating_test_types(self, fp1: Dict[str, Any], fp2: Dict[str, Any]) -> List[str]:
    """Identify test types that best differentiate between two models."""
    test_types = ["random", "powers_of_10", "repeated_digits", "near_powers_of_10",
"repeating_patterns"]
    differentiating_types = []

    # Compare test type performance if available
    tp1 = fp1.get('test_type_performance', {})
    tp2 = fp2.get('test_type_performance', {})

    for test_type in test_types:
        if test_type in tp1 and test_type in tp2:
            # Check if there's a significant difference in accuracy
            acc1 = tp1[test_type].get('accuracy', 0)
            acc2 = tp2[test_type].get('accuracy', 0)
```

```
        if abs(acc1 - acc2) > 0.1:  # Significant difference threshold
            differentiating_types.append(test_type)
            continue
```

This targeted approach analyzes existing fingerprints to identify which specific test types and digit lengths provide maximum differentiation between competing models.

3.3 Fingerprint Comparison and Model Identification

To identify an unknown model, I compare its fingerprint against a database of known fingerprints using a weighted similarity calculation:

```
# From fingerprint_module.py
def _calculate_similarity(self, fp1: ModelFingerprint, fp2: ModelFingerprint) -> Tuple[float,
Dict[str, float]]:
    """Calculate similarity score between two fingerprints."""
    components = {}

    # 1. Compare error growth rates
    if (fp1.fingerprint.get('error_growth_rate') is not None and
        fp2.fingerprint.get('error_growth_rate') is not None):

        fp1_growth = fp1.fingerprint['error_growth_rate']
        fp2_growth = fp2.fingerprint['error_growth_rate']

        if fp2_growth > 0:
            growth_similarity = 1 - min(abs(fp1_growth - fp2_growth) / fp2_growth, 1)
            components['growth_rate'] = growth_similarity

    # 2. Compare direction bias
    if (fp1.fingerprint.get('direction_bias') is not None and
        fp2.fingerprint.get('direction_bias') is not None):

        fp1_bias = fp1.fingerprint['direction_bias']
        fp2_bias = fp2.fingerprint['direction_bias']

        # Check if sign matches
        sign_match = np.sign(fp1_bias) == np.sign(fp2_bias)

        # Normalize the magnitude difference
        bias_similarity = 1 - min(abs(fp1_bias - fp2_bias) / (abs(fp2_bias) + 0.01), 1)

        # Adjust similarity based on sign match
```

```
        bias_similarity = bias_similarity * 0.5 + (0.5 if sign_match else 0)
        components['direction_bias'] = bias_similarity

    # Calculate weighted score
    weights = {
        'growth_rate': 0.3,
        'direction_bias': 0.2,
        'error_pattern': 0.3,
        'special_types': 0.1,
        'digit_positions': 0.1
    }

    # Only consider components that exist
    available_weights = {comp: weight for comp, weight in weights.items() if comp in
components}

    total_weight = sum(available_weights.values())

    if total_weight > 0:
        # Normalize weights
        norm_weights = {comp: weight/total_weight for comp, weight in available_weights.items()}

        # Calculate weighted score
        score = sum(components[comp] * norm_weights[comp] for comp in components.keys())
    else:
        score = 0

    return score, components
```

The system calculates similarity scores across multiple fingerprint components, each weighted according to its discriminative power:

- Growth rate (30%)
- Direction bias (20%)
- Error pattern (30%)
- Special test type performance (10%)
- Digit position errors (10%)

3.4 Empirical Validation and Performance Metrics

To validate the approach, I conducted testing to assess the system's performance across multiple models. These tests were designed to evaluate both identification accuracy and efficiency.

3.4.1 Llama 3.2 (1B) Confidence Tests

I ran 10 identification sessions against Llama 3.2 (1B) with the following results:


```
// Sample test result from confidence_test_llama3.2_1b_20250318_144935.json
{
  "run": 8,
  "identified_model": "llama3.2:1b",
  "correct_identification": true,
  "confidence": 0.8678053181905571,
  "tests_run": 10,
  "api_calls": 7,
  "duration_seconds": 14.516581535339355,
  "top_matches": [
    {
      "model": "llama3.2:1b",
      "score": 0.8678053181905571
    },
    {
      "model": "gemma3:1b",
      "score": 0.14323296588706338
    },
    {
      "model": "phi4-mini:latest",
      "score": 0.06481490647109985
    }
  ],
  "stage_1_confidence": 0.8678053181905571,
  "stage_1_tests": 10
}
```


Summarizing all 10 runs for Llama 3.2:

- **Accuracy**: 90% correct identification rate (9/10 runs)
- **Confidence**: Average confidence score of 53.8%
- **Efficiency**: Average of 8.7 API calls per identification
- **Speed**: Average identification time of 18.3 seconds
- **First-stage success**: 70% of identifications completed with only the initial screening tests

The single misidentification occurred in run #4, where the system incorrectly identified the model as Gemma 3 (1B) with a confidence of 44.4%.

3.4.2 Gemma 3 (1B) Confidence Tests

I conducted similar tests on Gemma 3 (1B) to evaluate the system's performance on a different architecture:


```
// Sample test result from confidence_test_gemma3_1b_20250318_162440.json
{
  "run": 4,
  "identified_model": "gemma3:1b",
  "correct_identification": true,
  "confidence": 0.8903034384875314,
  "tests_run": 10,
  "api_calls": 8,
  "duration_seconds": 16.994433879852295,
  "top_matches": [
    {
      "model": "gemma3:1b",
      "score": 0.8903034384875314
    },
    {
      "model": "phi4-mini:latest",
      "score": 0.25347045832738535
    },
    {
      "model": "llama3.2:1b",
      "score": 0.06667067398984029
    }
  ],
  "stage_1_confidence": 0.8903034384875314,
  "stage_1_tests": 10
}
```


Summarizing all 10 runs for Gemma 3:

- **Accuracy**: 80% correct identification rate (8/10 runs)
- **Confidence**: Average confidence score of 55.7%
- **Efficiency**: Average of 11.9 API calls per identification
- **Speed**: Average identification time of 25.7 seconds
- **Second-stage requirement**: 60% of identifications required targeted testing beyond the initial screening

The two misidentifications classified the model as Phi-4 Mini and Llama 3.2, suggesting some overlap in error patterns between these architecturally similar models.

3.4.3 Phi-4 Mini Confidence Tests

Testing on Phi-4 Mini revealed interesting challenges in differentiation:


```
// Sample test result from confidence_test_phi4-mini_latest_20250318_182512.json
{
  "run": 7,
  "identified_model": "phi4-mini:latest",
  "correct_identification": true,
  "confidence": 0.45307090779250553,
  "tests_run": 15,
  "api_calls": 11,
  "duration_seconds": 23.606383800506592,
  "top_matches": [
    {
      "model": "phi4-mini:latest",
      "score": 0.45307090779250553
    },
    {
      "model": "gemma3:1b",
      "score": 0.28559229905286637
    },
    {
      "model": "llama3.2:1b",
      "score": 0.07766811678831419
    }
  ],
  "stage_1_confidence": 0.24715163737464937,
  "stage_1_tests": 10,
  "stage_2_confidence": 0.45307090779250553,
  "stage_2_tests": 5
}
```


Summarizing all 10 runs for Phi-4 Mini:

- **Accuracy**: 40% correct identification rate (4/10 runs)
- **Misidentification pattern**: 40% identified as Llama 3.2, 20% as Gemma 3
- **Confidence**: Average confidence score of 38.9%
- **Efficiency**: Average of 8.1 API calls per identification
- **Speed**: Average identification time of 19.3 seconds
- **Extended testing**: 40% of runs required targeted testing beyond initial screening

These results indicate that Phi-4 Mini presents greater identification challenges compared to the other tested models. The lower identification accuracy suggests that its mathematical error patterns may be less distinctive or may exhibit higher variability between test sessions. This could be due to factors such as its training methodology, internal representation of numerical

operations, or how the model's architecture handles computational tasks. Further investigation with expanded test batteries might be needed to develop more reliable fingerprinting for this particular model.

3.4.4 Overall Performance Analysis

Combining all test runs across the three models:

- **Overall accuracy**: 70% (21/30 correct identifications)
- **Average API calls**: 9.6 calls per identification
- **Average runtime**: 21.13 seconds per identification
- **First-stage success rate**: 56.7% completed with just screening tests
- **Confidence correlation**: Higher confidence scores (>65%) correlated with 91.7% correct identifications

These results demonstrate that the mathematical error fingerprinting approach provides an effective and efficient method for model identification. The method works particularly well for models with distinctive error signatures (like Llama 3.2), while architecturally similar models (like Gemma 3 and Phi-4 Mini) present greater classification challenges, sometimes requiring additional targeted testing.

4. Results

4.1 Distinctive Error Fingerprints

My analysis supports that each model produces unique and consistent mathematical error patterns:

1. **Model-Specific Error Distributions**: As shown in my test reports, each model demonstrated distinctive error rates across digit lengths. For example:
    - Llama 3.2 (1B) showed a distinctive negative direction bias (-0.450) and relatively consistent error rates across digit lengths
    - Gemma 3 (1B) displayed a positive direction bias (0.533) and dramatic error spikes at specific digit thresholds
    - Phi-4 Mini exhibited unique performance on repeated digit tests (60% accuracy) compared to other models
2. **Fingerprint Stability**: Error patterns remained consistent across multiple test runs, indicating stable model behaviors rather than random variation.

3. **Cross-Model Differentiation**: Even models with similar architectures and parameter counts showed clearly distinguishable error signatures.

4.2 Identification System Performance

My fingerprinting system demonstrated mixed capabilities for model identification, with performance varying significantly across different models:

Variable Identification Accuracy: The overall accuracy across all test models was 70%, with significant variation between models:

- Llama 3.2 (1B): 90% correct identification rate
- Gemma 3 (1B): 80% correct identification rate
- Phi-4 Mini: 40% correct identification rate

1. **Model-Dependent Performance**: The system performed well on architecturally distinctive models but struggled with differentiating between models with similar error patterns. The consistent misidentification of Phi-4 Mini as Gemma 3 indicates a limitation in the current fingerprinting approach for certain model families.

2. **Confidence Correlation:** Higher confidence scores generally correlated with correct identifications. When the system reported confidence scores above 65%, it achieved 83.3% accuracy (5/6 tests), suggesting that confidence thresholds could be used to increase reliability.
3. **Efficient Test Requirements**: Most models could be evaluated with reasonable efficiency (average of 9.6 API calls), with over half of the identifications completed using just the initial 10 screening tests.
4. **Discriminative Test Types:** Analysis revealed that edge case tests provided the strongest differentiation between models (with up to 60% accuracy differences), followed by repeated digits tests and powers of 10 tests. Edge cases were particularly effective at distinguishing Llama 3.2 from other models, while powers of 10 tests helped differentiate Phi-4 Mini from Gemma 3.

4.3 Experimental Validation on Consumer Hardware

I validated my approach through controlled experiments conducted on personal consumer hardware, with important constraints and limitations:

1. **Identification of Llama 3.2 (1B)**: My system identified this model with 65.37% confidence after just 10 tests, correctly distinguishing it from similar models. This testing was performed on a standard desktop running a local Ollama instance, demonstrating

the approach's viability even with limited computational resources.

2. **Targeted Testing Effectiveness**: Testing on Phi-4 Mini revealed that adding targeted tests beyond the initial screening improved identification accuracy from 0% (with just screening tests) to 66.7% (with targeted tests). The most successful identification approach used 15 tests (5 screening + 10 targeted), achieving 80% accuracy. This demonstrates how targeted testing can significantly enhance differentiation between models with similar error patterns.

Limitations and Assumptions

These experiments operated under several key constraints that would affect real-world applications:

1. **Limited Model Set**: Testing was restricted to small-parameter open-source models (1B-7B parameters) runnable on consumer hardware, whereas commercial applications often use much larger models (70B-1T parameters).

2. **Controlled Environment**: All tests used identical hardware, software versions, and controlled temperature conditions. In real-world settings, varying deployment environments could impact response patterns.

3. **Same Interface Assumption**: The approach assumes API interfaces that accept direct mathematical queries without formatting restrictions or guardrails that might reject or modify simple calculation prompts.

4. **No Anti-Fingerprinting Measures**: These tests assume no deliberate countermeasures are in place to obscure mathematical error patterns, which sophisticated providers might implement.

5. **Computational Overhead**: Even with optimization, identification requires 8-15 API calls, which may incur costs or rate limits when working with commercial services.

For real-world deployment, the methodology would need expansion to include more diverse models, adaptation to various API constraints, and strategies to overcome potential anti-fingerprinting measures that might be implemented by providers seeking to prevent model identification.

5. Applications

5.1 Model Verification and Transparency

My mathematical fingerprinting methodology offers a practical tool for enhancing transparency in AI services. As LLMs become more widespread in commercial applications, organizations and end-users can use this approach to test which model is actually being used by a service, regardless of what is claimed. This independent verification capability is particularly valuable in contexts where model capabilities directly impact service quality, security, and user experience.

5.2 Security Applications

The fingerprinting approach has several security applications:

1. **Attribution**: Identifying which model was used to generate specific content by analyzing error patterns in any included calculations.

2. **Model-Specific Vulnerability Assessment**: Once a specific model is identified, targeted jailbreaking techniques that exploit known vulnerabilities in that particular model can potentially be applied. Different models have different safety guardrails, training processes, and filtering mechanisms, making them susceptible to different attack. Accurately identifying the model becomes the crucial first step in assessing and potentially exploiting model-specific vulnerabilities.

5.3 Research Applications

Beyond its practical applications, my fingerprinting methodology provides tools for LLM research:

1. **Tracking Model Evolution**: My analysis revealed distinctive fingerprints between different model families (e.g., Llama vs. Gemma). For example, Llama 3.2 (1B) showed a consistent negative direction bias (-0.450) across calculations, while Gemma 3 (1B) exhibited a positive direction bias (0.533). These fundamental differences could serve as baselines to track how subsequent versions of these models evolve in their handling of mathematical operations. Changes in these patterns could reveal how architectural modifications or training approaches affect numerical reasoning capabilities.
2. **Feature Representation Study**: The fingerprints revealed specific patterns in how models handle different types of calculations. For instance, my experiments showed that Phi-4 Mini achieved 60% accuracy on repeated digits tests compared to 0% for Llama 3.2, suggesting fundamentally different internal representations of repeating numerical patterns. These distinct error signatures provide a view into how each architecture encodes and processes mathematical knowledge.
3. **Insights into Model Limitations:** My data revealed distinct performance cliffs where error rates suddenly increased at specific digit thresholds. Gemma 3 showed a 5902% error increase between 2-3 digits and another 689% jump between 4-5 digits. Phi-4 Mini similarly demonstrated extreme error spikes at specific transition points. These abrupt threshold effects suggest fundamental limitations in how these models represent and

process numbers. The non-linear relationship between problem complexity and error rates points to architectural constraints, possibly related to positional encoding schemes or internal numerical representations, rather than simple computational scaling limitations.

6. Discussion

6.1 From Mathematical Errors to Model Identification

Error patterns in mathematical computation reveal broader implications for how we evaluate and understand LLMs. Rather than treating calculation errors as simple deficiencies, they can be more productively viewed as windows into internal model mechanics. This perspective suggests that developing specialized probes for different cognitive tasks could yield similar identification signatures, potentially offering more nuanced evaluation metrics than traditional benchmarks. The methodology developed in this research demonstrates how targeted testing focused on fundamental operations can efficiently extract meaningful model characteristics without requiring exhaustive evaluation across traditional benchmark suites.

6.2 Theoretical Implications for Knowledge Representation

My findings provide insights into how mathematical knowledge is represented within LLMs:

1. **Superposition Hypothesis**: The distinctive error patterns I observed support the theory that mathematical facts and procedures are stored in superposition within model parameters. As suggested by research on toy models of superposition (Elhage et al., 2022), there are more features represented than available dimensions, leading to interference patterns that manifest as systematic errors.

2. **Capacity Boundaries**: The sharp increases in error rates at specific complexity thresholds suggest fundamental limitations in how models represent numerical information, potentially related to context window constraints or attention mechanism limitations.

3. **Architectural Fingerprints**: The unique error signatures of different models likely reflect their architectural differences, training data variations, and optimization approaches.

6.3 Practical Applications and Deployment

The mathematical fingerprinting approach I developed has several immediate applications:

1. **API Service Verification**: Organizations can verify that they are accessing the claimed model in commercial API services, ensuring they receive the capabilities they're paying for.

2. **Model Substitution Detection**: Regulatory bodies and researchers can detect "bait and switch" practices where lower-quality models are substituted for higher-quality ones.
3. **Unauthorized Use Detection**: Original model creators can potentially detect unauthorized commercial deployment of their models, helping protect intellectual property rights and ensure compliance with licensing terms.

6.4 Future Research Directions

This work opens several promising research directions:

1. **Expanded Model Coverage**: Extending my fingerprinting database to cover more commercial and open-source models.

2. **Cross-Modal Fingerprinting**: Investigating whether similar distinctive patterns exist for other tasks beyond mathematics.

3. **Longitudinal Stability**: Studying how fingerprints evolve across model versions and fine-tunings.

7. Conclusion

What began as an investigation into mathematical errors in LLMs led to a novel and practical approach for model identification. My mathematical error fingerprinting methodology provides both a useful tool for ensuring transparency in AI deployments and insights into how these models represent and process mathematical knowledge.

While the current implementation demonstrates promising results for distinguishing models with distinctive error patterns like Llama 3.2, the challenges in differentiating between architecturally similar models like Gemma 3 and Phi-4 Mini highlight areas for future improvement. By refining the fingerprinting techniques and expanding the test battery, the methodology could potentially achieve higher accuracy across a broader range of models.

References

1. Bukatin, Michael, and Jon Anthony. "Dataflow Matrix Machines as a Model of Computations with Linear Streams." arXiv preprint arXiv:1706.00648 (2017).https://arxiv.org/pdf/1706.00648 Accessed: 2025-02-06 (PEER REVIEWED ARTICLE)
2. Dubey, Abhimanyu, et al. "The llama 3 herd of models." arXiv preprint arXiv:2407.21783 (2024).https://arxiv.org/pdf/2407.21783 Accessed: 2025-02-03 (RESEARCH PAPER)
3. Palamas, Theodoros. Investigating the ability of neural networks to learn simple modular arithmetic. Diss. Master's thesis, The University of Edinburgh,

2017.https://project-archive.inf.ed.ac.uk/msc/20172390/msc_proj.pdf Accessed: 2025-02-05 (Dissertation)
4. Fawzi, Alhussein, et al. "Discovering faster matrix multiplication algorithms with reinforcement learning." Nature 610.7930 (2022): 47-53. Accessed: 2024-02-05 (PEER REVIED ARTICLE)
5. Siegelmann, Hava T., and Eduardo D. Sontag. "On the computational power of neural nets." Proceedings of the fifth annual workshop on Computational learning theory. 1992. https://binds.cs.umass.edu/papers/1992_Siegelmann_COLT.pdf Accessed: 2025-02-05 (PEER REVIEWED ARTICLE)
6. Vaswani, Ashish, et al. "Attention is all you need." Advances in neural information processing systems 30 (2017). Accessed: 2024-02-23 (PEER REVIEWED ARTICLE) https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf
7. Tao, Terence “Terence Tao - Machine-Assisted Proofs (February 19, 2025)” Youtube, 2025, Accessed: 2025-02-20 (VIDEO LECTURE)
Terence Tao - Machine-Assisted Proofs (February 19, 2025)
8. Sanderson, Grant “How might LLMs store facts | DL7” Youtube, 2024, Accessed: 2025-02-23https://www.youtube.com/watch?v=9-Jl0dxWQs8&list=PLFF7OyfORULjHKD18YK9gmNYAsARIfwBV&index=1&t=1s
9. Neel. “Neel Nanda.” Neel Nanda, 21 Dec. 2022, https://www.neelnanda.io/mechanistic-interpretability/glossary Accessed: 2015-02-25 (Research Guide)

Appendices

Appendix A: Detailed Error Analysis

A.1 Llama 3.2 (1B) Error Profile

Digit Length 2: 174.85% error

Digit Length 3: 78.29% error

Digit Length 4: 82.66% error

Digit Length 5: 128.14% error

Digit Length 6: 163.67% error

Digit Length 7: 71.23% error

Digit Length 8: 113.31% error

Digit Length 9: 99.93% error

Direction Bias: -0.450 (Model tends to underestimate results)

Performance on Test Types:

- Random: 0.00% accuracy, 89.48% avg error
- Edge Case: 0.00% accuracy, 100.50% avg error
- Repeated Digits: 0.00% accuracy, 89.27% avg error
- Near Powers Of 10: 0.00% accuracy, 234.18% avg error
- Powers Of 10: 20.00% accuracy, 217.40% avg error
- Repeating Patterns: 0.00% accuracy, 94.09% avg error

A.2 Gemma 3 (1B) Error Profile

Digit Length 2: 13.79% error

Digit Length 3: 827.91% error

Digit Length 4: 237.20% error

Digit Length 5: 1873.49% error

Digit Length 6: 1003.69% error

Digit Length 7: 395.16% error

Digit Length 8: 458.93% error

Direction Bias: 0.533 (Model tends to overestimate results)

Performance on Test Types:

- Random: 2.86% accuracy, 831.47% avg error
- Repeating Patterns: 0.00% accuracy, 596.86% avg error
- Near Powers Of 10: 20.00% accuracy, 60.12% avg error
- Repeated Digits: 40.00% accuracy, 179.58% avg error
- Powers Of 10: 20.00% accuracy, 720.00% avg error
- Edge Case: 60.00% accuracy, 2020.04% avg error

A.3 Phi-4 Mini Error Profile

Digit Length 1: 0.00% error

Digit Length 2: 172.19% error

Digit Length 3: 1659335100.35% error

Digit Length 4: 84.96% error

Digit Length 5: 234.69% error

Digit Length 6: 3057.93% error

Digit Length 7: 181.17% error

Digit Length 8: 201.77% error

Digit Length 9: 9915.49% error

Direction Bias: 0.117 (Slight tendency to overestimate)

Performance on Test Types:

- Random: 5.71% accuracy, 489.55% avg error
- Repeated Digits: 60.00% accuracy, 2986803006.29% avg error
- Powers Of 10: 60.00% accuracy, 360.00% avg error
- Near Powers Of 10: 0.00% accuracy, 2.60% avg error
- Repeating Patterns: 0.00% accuracy, 4062.14% avg error
- Edge Case: 100.00% accuracy, 0.00% avg error

Appendix B: Fingerprinting Methodology

B.1 Test Case Generation Details

The test cases used in this study were designed to cover a range of mathematical patterns that might reveal distinctive error behaviors:

1. **Standard Random Tests**: Multiplication problems with random numbers of specific digit lengths (2-9 digits)
2. **Powers of 10 Tests**: Problems involving powers of 10 (e.g., 1000 × 100)
3. **Repeated Digit Tests**: Problems with repeated digits (e.g., 777 × 777)
4. **Near Powers of 10 Tests**: Problems with numbers near powers of 10 (e.g., 999 × 99)
5. **Repeating Pattern Tests**: Problems with repeating patterns (e.g., 123123 × 456456)
6. **Edge Cases**: Special cases designed to stress computational boundaries (e.g., numbers with many carrying operations)

For screening tests, a balanced mix of these test types was used, with 10 tests total. For comprehensive testing, 5 tests from each category were used, for a total of 30 tests.

B.2 Fingerprint Component Details

The mathematical fingerprint consists of several key components:

1. **Error by Digit Length**: The average relative error for calculations of different digit lengths

2. **Error Growth Rate**: How quickly error grows as digit length increases
3. **Direction Bias**: Whether a model tends to over or underestimate results
4. **Test Type Performance**: Accuracy and average error for different test types
5. **Digit Error Patterns**: Analysis of which positions in the result are most likely to have errors

These components are combined using a weighted similarity calculation when comparing fingerprints, with the following weights:

- Growth rate: 30%
- Direction bias: 20%
- Error pattern: 30%
- Special test types: 10%
- Digit positions: 10%

B.3 Hierarchical Testing Approach

To minimize API calls, a two-stage testing approach is used:

1. **Initial Screening (10 tests)**: Quick assessment to identify potential matches
2. **Targeted Testing (5-15 additional tests)**: Only if needed, using tests specifically designed to differentiate between similar models

Tests in the second stage are selected based on analysis of the conflicting models' existing fingerprints, focusing on test types and digit lengths that show maximum differentiation.

Appendix C: Code and Resources

The complete implementation of the model identification system is available in the following files:

1. **api_client.py**: Handles interaction with model APIs, including request formatting, response parsing, and caching
2. **test_generator.py**: Generates test cases of various types
3. **fingerprint_module.py**: Creates and compares model fingerprints
4. **model_identifier.py**: Main identification logic
5. **test_confidence_script.py**: Use to run batch confidence test for any model

The database of model fingerprints and test results is stored in:

- **model_fingerprints.json**: Database of known model fingerprints
- **confidence_test_*.json**: Results of confidence tests for each model

All files can be found on Github: https://github.com/nmintzer-oswego/CSC322_Project_1.git