```
"""
cell1

Fixed Complete Stage 1 Experiment: Multi-dimensional attention analysis
解决负相关问题，优化特征权重和评分逻辑
"""
import torch
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy import stats
from transformers import AutoTokenizer, AutoModelForCausalLM
from typing import Dict, List


class EnhancedAttentionAnalyzer:
    def __init__(self):
        # 调整特征权重 - 增加熵的权重，减少其他特征权重
        self.feature_weights = {
            'entropy': 0.6,        # 主要特征：熵
            'variance': 0.2,       # 次要特征：方差
            'concentration': 0.1,  # 辅助特征：集中度
            'cross_layer': 0.1     # 辅助特征：跨层一致性
        }

    def extract_multi_dimensional_features(self, text, model, tokenizer):
        """提取多维度注意力特征"""
        inputs = tokenizer(text, return_tensors="pt", padding=True, truncation=True, max_length=512)

        with torch.no_grad():
            outputs = model(**inputs, output_attentions=True)

        # 获取所有层的注意力
        all_attentions = outputs.attentions

        features = {}

        # 1. 多层熵特征
        features.update(self.calculate_entropy_features(all_attentions, inputs))

        # 2. 注意力方差特征
        features.update(self.calculate_variance_features(all_attentions, inputs))

        # 3. 注意力集中度特征
        features.update(self.calculate_concentration_features(all_attentions, inputs))

        # 4. 跨层一致性特征
        features.update(self.calculate_cross_layer_features(all_attentions, inputs))

        return features

    def calculate_entropy_features(self, all_attentions, inputs):
        """计算多种熵特征"""
        seq_len = inputs['attention_mask'].sum().item()

        # 最后一层的熵
        last_layer = all_attentions[-1][0]  # [头数, seq_len, seq_len]
        last_token_entropy = self._calculate_token_entropy(
            last_layer[:, seq_len-1, :]
        )

        # 所有token的平均熵
        all_token_entropies = []
        for i in range(seq_len):
            token_entropy = self._calculate_token_entropy(last_layer[:, i, :])
            all_token_entropies.append(token_entropy)
        avg_token_entropy = np.mean(all_token_entropies)

        # 关键token的熵
        key_token_entropy = self._calculate_key_token_entropy(
            last_layer, inputs, seq_len
        )

        return {
            'last_token_entropy': last_token_entropy,
            'avg_token_entropy': avg_token_entropy,
```

```
            'key_token_entropy': key_token_entropy,
            'entropy_variance': np.var(all_token_entropies) if len(all_token_entropies) > 1 else 0
        }

    def calculate_variance_features(self, all_attentions, inputs):
        """计算注意力方差特征"""
        seq_len = inputs['attention_mask'].sum().item()
        last_layer = all_attentions[-1][0]

        # 每个头的注意力方差
        head_variances = []
        for head in range(last_layer.shape[0]):
            attn_weights = last_layer[head, seq_len-1, :seq_len]
            variance = torch.var(attn_weights).item()
            head_variances.append(variance)

        return {
            'avg_attention_variance': np.mean(head_variances),
            'max_attention_variance': np.max(head_variances),
            'variance_std': np.std(head_variances)
        }

    def calculate_concentration_features(self, all_attentions, inputs):
        """计算注意力集中度特征"""
        seq_len = inputs['attention_mask'].sum().item()
        last_layer = all_attentions[-1][0]

        max_attentions = []
        gini_coefficients = []

        for head in range(last_layer.shape[0]):
            attn_weights = last_layer[head, seq_len-1, :seq_len]

            # 集中度：最大注意力权重
            max_attention = torch.max(attn_weights).item()
            max_attentions.append(max_attention)

            # Gini系数（不平等程度）
            gini = self._calculate_gini_coefficient(attn_weights)
            gini_coefficients.append(gini)

        return {
            'avg_max_attention': np.mean(max_attentions),
            'avg_gini_coefficient': np.mean(gini_coefficients),
        }

    def calculate_cross_layer_features(self, all_attentions, inputs):
        """计算跨层一致性特征"""
        seq_len = inputs['attention_mask'].sum().item()

        # 比较最后3层的注意力模式相似度
        if len(all_attentions) >= 3:
            last_layers = all_attentions[-3:]
            correlations = []

            for i in range(len(last_layers)-1):
                layer1 = last_layers[i][0][:, seq_len-1, :seq_len]
                layer2 = last_layers[i+1][0][:, seq_len-1, :seq_len]

                for head in range(layer1.shape[0]):
                    try:
                        # 确保数据是float类型
                        head1_data = layer1[head].float()
                        head2_data = layer2[head].float()

                        # 检查是否有足够的变异性
                        if torch.std(head1_data) > 1e-6 and torch.std(head2_data) > 1e-6:
                            corr_matrix = torch.corrcoef(torch.stack([head1_data, head2_data]))
                            corr = corr_matrix[0, 1]
                            if not torch.isnan(corr) and not torch.isinf(corr):
                                correlations.append(corr.item())
                    except:
                        continue

            cross_layer_consistency = np.mean(correlations) if correlations else 0
        else:
            cross_layer_consistency = 0
```

```
        return {
            'cross_layer_consistency': abs(cross_layer_consistency)  # 取绝对值，关注一致性强度
        }

    def _calculate_token_entropy(self, attention_weights):
        """计算单个token的平均熵"""
        entropies = []
        for head_attn in attention_weights:
            head_attn = head_attn + 1e-9
            entropy = -torch.sum(head_attn * torch.log(head_attn))
            entropies.append(entropy.item())
        return np.mean(entropies)

    def _calculate_key_token_entropy(self, attention_matrix, inputs, seq_len):
        """计算关键token的熵"""
        if seq_len > 3:
            middle_start = 1
            middle_end = seq_len - 1
            middle_entropies = []

            for i in range(middle_start, middle_end):
                token_entropy = self._calculate_token_entropy(attention_matrix[:, i, :])
                middle_entropies.append(token_entropy)

            return np.mean(middle_entropies) if middle_entropies else 0
        else:
            return self._calculate_token_entropy(attention_matrix[:, seq_len-1, :])

    def _calculate_gini_coefficient(self, weights):
        """计算Gini系数（衡量不平等程度）"""
        try:
            weights = weights.detach().cpu().numpy()
            weights = np.sort(weights)
            n = len(weights)
            if n == 0 or weights.sum() == 0:
                return 0

            cumsum = np.cumsum(weights)
            gini = (n + 1 - 2 * np.sum(cumsum) / cumsum[-1]) / n
            return max(0, min(1, gini))  # 确保在[0,1]范围内
        except:
            return 0

    def predict_complexity_enhanced(self, features):
        """基于多维特征预测复杂度 - 修复后版本"""

        # 更稳健的特征归一化
        entropy_score = self._robust_normalize(features['avg_token_entropy'], 0.5, 2.5)
        variance_score = self._robust_normalize(features['avg_attention_variance'], 0, 0.2)
        concentration_score = self._robust_normalize(features['avg_max_attention'], 0.2, 0.8)
        consistency_score = self._robust_normalize(features['cross_layer_consistency'], 0, 1)

        # 加权组合 - 注意方向
        complexity_score = (
            self.feature_weights['entropy'] * entropy_score +           # 熵越高越复杂
            self.feature_weights['variance'] * variance_score +         # 方差越高越复杂
            self.feature_weights['concentration'] * (1 - concentration_score) +  # 集中度越低(分散)越复杂
            self.feature_weights['cross_layer'] * consistency_score     # 一致性越高越复杂
        )

        complexity_score = max(0, min(1, complexity_score))  # 确保在[0,1]范围内

        return {
            'complexity_score': complexity_score,
            'features': features,
            'is_complex': complexity_score > 0.5,
            'avg_entropy': features['avg_token_entropy'],  # 兼容性
            'head_entropies': [features['avg_token_entropy']] * 8,  # 兼容性
            'feature_breakdown': {
                'entropy_score': entropy_score,
                'variance_score': variance_score,
                'concentration_score': concentration_score,
                'consistency_score': consistency_score
            }
        }
```

```
    def _robust_normalize(self, value, min_val, max_val, target_min=0, target_max=1):
        """更稳健的归一化函数"""
        if max_val <= min_val:
            return target_min

        # 线性映射到目标范围
        normalized = (value - min_val) / (max_val - min_val)
        normalized = max(0, min(1, normalized))  # 截断到[0,1]

        # 映射到目标范围
        return target_min + normalized * (target_max - target_min)

    # 保持兼容性的简化方法
    def predict_complexity(self, attention_weights):
        """兼容原始方法的简化版本"""
        entropies = []
        for head_attn in attention_weights:
            head_attn = head_attn + 1e-9
            entropy = -torch.sum(head_attn * torch.log(head_attn))
            entropies.append(entropy.item())

        avg_entropy = np.mean(entropies)
        complexity_score = self._robust_normalize(avg_entropy, 0.5, 2.5)

        return {
            'complexity_score': complexity_score,
            'avg_entropy': avg_entropy,
            'head_entropies': entropies,
            'is_complex': complexity_score > 0.5
        }


class ComprehensiveBaselineExperiment:
    def __init__(self, model_name="microsoft/DialoGPT-small", use_enhanced=False):
        self.tokenizer = AutoTokenizer.from_pretrained(model_name)
        self.model = AutoModelForCausalLM.from_pretrained(
            model_name,
            output_attentions=True,
            attn_implementation="eager"
        )
        self.analyzer = EnhancedAttentionAnalyzer()
        self.use_enhanced = use_enhanced

        if self.tokenizer.pad_token is None:
            self.tokenizer.pad_token = self.tokenizer.eos_token

        # 优化后的测试数据集 - 更清晰的复杂度分级
        self.task_dataset = {
            "simple": [
                "What is 2+2?",
                "The capital of France is Paris",
                "My name is John",
                "What color is the sky?",
                "How many days in a week?",
                "What is water made of?",
                "The sun rises in the east",
                "1+1 equals 2",
                "Cats are animals",
                "The alphabet starts with A"
            ],
            "medium": [
                "Why do objects fall down?",
                "How does a bicycle work?",
                "What causes rain to fall?",
                "Why is the ocean salty?",
                "How do plants make their food?",
                "What makes ice float on water?",
                "Why do we have different seasons?",
                "How do computers process information?",
                "What is electricity and how it works?",
                "Why do people dream during sleep?"
            ],
            "complex": [
                "Explain the relationship between quantum mechanics and general relativity",
                "Analyze the economic impact of artificial intelligence on employment markets",
                "What would happen if gravity suddenly became twice as strong?",
                "Discuss the ethical implications of genetic engineering in humans",
```

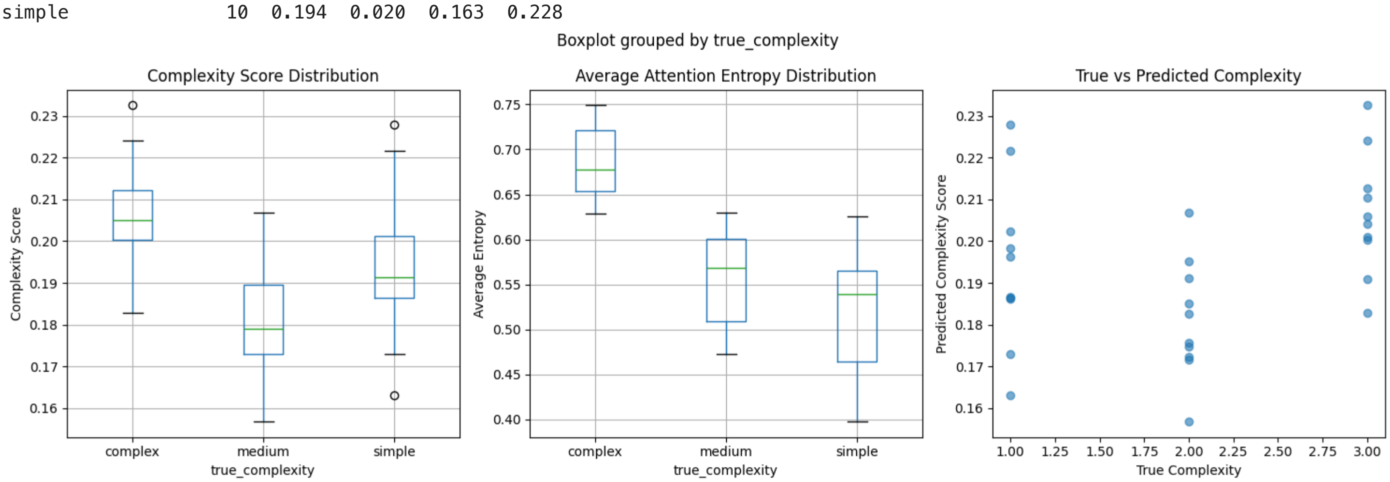
```
            "How might climate change affect global food security systems?",
            "Evaluate the societal effects of social media on democratic processes",
            "Compare the advantages and disadvantages of renewable energy sources",
            "How do cultural differences affect international business negotiations?",
            "What are the long-term consequences of space exploration for humanity?",
            "Analyze the philosophical implications of consciousness in artificial intelligence"
        ]
    }

def extract_attention_features(self, text, method="last_token"):
    """Extract attention features"""
    inputs = self.tokenizer(text, return_tensors="pt", padding=True, truncation=True, max_length=512)

    with torch.no_grad():
        outputs = self.model(**inputs, output_attentions=True)

    last_attention = outputs.attentions[-1][0]

    if method == "last_token":
        seq_len = inputs['attention_mask'].sum().item()
        last_token_attn = last_attention[:, seq_len-1, :]
        return last_token_attn
    elif method == "average":
        return last_attention
    else:
        raise ValueError(f"Unknown method: {method}")

def run_single_task(self, text, complexity_label):
    """Run single task with option for enhanced or basic analysis"""
    if self.use_enhanced:
        features = self.analyzer.extract_multi_dimensional_features(text, self.model, self.tokenizer)
        result = self.analyzer.predict_complexity_enhanced(features)
    else:
        attention_weights = self.extract_attention_features(text)
        result = self.analyzer.predict_complexity(attention_weights)

    result['true_complexity'] = complexity_label
    result['task'] = text

    return result

def run_comprehensive_experiment(self):
    """Run comprehensive experiment"""
    method_name = "Enhanced Multi-dimensional" if self.use_enhanced else "Basic Entropy"
    print(f"🧪 Running {method_name} Stage 1 experiment...")

    all_results = []

    for complexity, tasks in self.task_dataset.items():
        print(f"\n📊 Processing {complexity} tasks...")

        for task in tasks:
            result = self.run_single_task(task, complexity)
            all_results.append(result)
            print(f"'{task[:50]}...' -> complexity={result['complexity_score']:.3f}")

    df = pd.DataFrame(all_results)
    return self.analyze_results(df)

def analyze_results(self, df):
    """Comprehensive result analysis"""
    method_name = "Enhanced Multi-dimensional" if self.use_enhanced else "Basic Entropy"
    print("\n" + "="*60)
    print(f"📈 {method_name.upper()} RESULTS ANALYSIS")
    print("="*60)

    # 1. Descriptive statistics
    summary = df.groupby('true_complexity')['complexity_score'].agg([
        'count', 'mean', 'std', 'min', 'max'
    ]).round(3)
    print("\n1. Descriptive Statistics:")
    print(summary)

    # 2. Visualization
    self.plot_results(df)

    # 3. Statistical tests
```

```
        self.statistical_tests(df)

        # 4. Correlation analysis
        self.correlation_analysis(df)

        # 5. Routing decision analysis
        self.routing_analysis(df)

        # 6. 如果是增强版，显示特征分解
        if self.use_enhanced and 'feature_breakdown' in df.iloc[0]:
            self.feature_breakdown_analysis(df)

        return df

    def plot_results(self, df):
        """Result visualization"""
        plt.figure(figsize=(15, 5))

        plt.subplot(1, 3, 1)
        df.boxplot(column='complexity_score', by='true_complexity', ax=plt.gca())
        plt.title('Complexity Score Distribution')
        plt.ylabel('Complexity Score')

        plt.subplot(1, 3, 2)
        df.boxplot(column='avg_entropy', by='true_complexity', ax=plt.gca())
        plt.title('Average Attention Entropy Distribution')
        plt.ylabel('Average Entropy')

        plt.subplot(1, 3, 3)
        complexity_mapping = {'simple': 1, 'medium': 2, 'complex': 3}
        df['complexity_numeric'] = df['true_complexity'].map(complexity_mapping)
        plt.scatter(df['complexity_numeric'], df['complexity_score'], alpha=0.6)
        plt.xlabel('True Complexity')
        plt.ylabel('Predicted Complexity Score')
        plt.title('True vs Predicted Complexity')

        plt.tight_layout()
        plt.show()

    def statistical_tests(self, df):
        """Statistical significance testing"""
        print("\n2. Statistical Significance Tests:")

        simple_scores = df[df['true_complexity'] == 'simple']['complexity_score']
        medium_scores = df[df['true_complexity'] == 'medium']['complexity_score']
        complex_scores = df[df['true_complexity'] == 'complex']['complexity_score']

        f_stat, p_value = stats.f_oneway(simple_scores, medium_scores, complex_scores)
        print(f"ANOVA F-statistic: {f_stat:.4f}, p-value: {p_value:.4f}")

        if p_value < 0.05:
            print("✅ Significant difference between groups (p < 0.05)")
        else:
            print("❌ No significant difference between groups (p >= 0.05)")

        from scipy.stats import ttest_ind
        t1, p1 = ttest_ind(simple_scores, complex_scores)
        print(f"Simple vs Complex tasks t-test: t={t1:.3f}, p={p1:.4f}")

    def correlation_analysis(self, df):
        """Correlation analysis"""
        print("\n3. Correlation Analysis:")

        complexity_mapping = {'simple': 1, 'medium': 2, 'complex': 3}
        df['complexity_numeric'] = df['true_complexity'].map(complexity_mapping)

        correlation = df['complexity_score'].corr(df['complexity_numeric'])
        print(f"Correlation between complexity score and true complexity: {correlation:.4f}")

        if correlation > 0.5:
            print("✅ Strong positive correlation - Hypothesis validated!")
        elif correlation > 0.3:
            print("⚠️ Moderate correlation - Some effectiveness but needs improvement")
        elif correlation < -0.5:
            print("❌ Strong negative correlation - Feature direction needs fixing")
        elif correlation < -0.3:
            print("⚠️ Moderate negative correlation - Consider reversing score logic")
```

```
        else:
            print("❌ Weak correlation - Need to reconsider methodology")

    def routing_analysis(self, df):
        """Routing decision analysis"""
        print("\n4. Routing Decision Analysis:")

        routing_stats = df.groupby('true_complexity')['is_complex'].agg([
            'count', 'sum', lambda x: (x.sum() / len(x) * 100)
        ]).round(1)
        routing_stats.columns = ['Total', 'Routed to Cloud', 'Routing Rate (%)']
        print(routing_stats)

        simple_correct = (df[df['true_complexity'] == 'simple']['is_complex'] == False).sum()
        complex_correct = (df[df['true_complexity'] == 'complex']['is_complex'] == True).sum()

        simple_total = len(df[df['true_complexity'] == 'simple'])
        complex_total = len(df[df['true_complexity'] == 'complex'])

        print(f"\nRouting Accuracy:")
        print(f"Simple task correct routing rate: {simple_correct/simple_total*100:.1f}%")
        print(f"Complex task correct routing rate: {complex_correct/complex_total*100:.1f}%")

    def feature_breakdown_analysis(self, df):
        """分析各个特征的贡献"""
        print("\n5. Feature Breakdown Analysis:")

        if 'feature_breakdown' in df.iloc[0]:
            feature_data = []
            for _, row in df.iterrows():
                breakdown = row['feature_breakdown']
                feature_data.append({
                    'complexity': row['true_complexity'],
                    'entropy_score': breakdown['entropy_score'],
                    'variance_score': breakdown['variance_score'],
                    'concentration_score': breakdown['concentration_score'],
                    'consistency_score': breakdown['consistency_score']
                })

            feature_df = pd.DataFrame(feature_data)
            feature_summary = feature_df.groupby('complexity')[
                ['entropy_score', 'variance_score', 'concentration_score', 'consistency_score']
            ].mean().round(3)

            print("Average feature scores by complexity:")
            print(feature_summary)

    def save_results(self, df, filename="stage1_results.csv"):
        """Save results"""
        df.to_csv(filename, index=False)
        print(f"\n💾 Results saved to {filename}")


# Colab友好的运行函数
def quick_run_enhanced():
    """快速运行增强版实验"""
    experiment = ComprehensiveBaselineExperiment(use_enhanced=True)
    results_df = experiment.run_comprehensive_experiment()
    experiment.save_results(results_df, "enhanced_results_fixed.csv")
    return results_df

def quick_run_basic():
    """快速运行基础版实验"""
    experiment = ComprehensiveBaselineExperiment(use_enhanced=False)
    results_df = experiment.run_comprehensive_experiment()
    experiment.save_results(results_df, "basic_results_fixed.csv")
    return results_df

def run_comparison_experiment():
    """运行对比实验"""
    print("🔬 Running Comparison Experiment: Basic vs Enhanced (Fixed)")
    print("="*60)

    print("\n🔵 Running Basic Entropy Method...")
    experiment_basic = ComprehensiveBaselineExperiment(use_enhanced=False)
    results_basic = experiment_basic.run_comprehensive_experiment()
```

```
    print("\n" + "="*60)

    print("\n🟢 Running Enhanced Multi-dimensional Method (Fixed)...")
    experiment_enhanced = ComprehensiveBaselineExperiment(use_enhanced=True)
    results_enhanced = experiment_enhanced.run_comprehensive_experiment()

    experiment_basic.save_results(results_basic, "basic_method_fixed.csv")
    experiment_enhanced.save_results(results_enhanced, "enhanced_method_fixed.csv")

    return results_basic, results_enhanced

# 自动运行
if __name__ == "__main__":
    print("🎯 Auto-running Enhanced Multi-dimensional Method (Fixed Version)...")
    results = quick_run_enhanced()
    print("\n🎉 Stage 1 experiment completed!")
```

```
🎯 Auto-running Enhanced Multi-dimensional Method (Fixed Version)...
The following generation flags are not valid and may be ignored: ['output_attentions']. Set `TRANSFORMERS_VERBOSITY=info` fo
The following generation flags are not valid and may be ignored: ['output_attentions']. Set `TRANSFORMERS_VERBOSITY=info` fo
🧪 Running Enhanced Multi-dimensional Stage 1 experiment...

📊 Processing simple tasks...
'What is 2+2?...' -> complexity=0.187
'The capital of France is Paris...' -> complexity=0.202
'My name is John...' -> complexity=0.228
'What color is the sky?...' -> complexity=0.163
'How many days in a week?...' -> complexity=0.198
'What is water made of?...' -> complexity=0.173
'The sun rises in the east...' -> complexity=0.196
'1+1 equals 2...' -> complexity=0.186
'Cats are animals...' -> complexity=0.222
'The alphabet starts with A...' -> complexity=0.186

📊 Processing medium tasks...
'Why do objects fall down?...' -> complexity=0.175
'How does a bicycle work?...' -> complexity=0.172
'What causes rain to fall?...' -> complexity=0.207
'Why is the ocean salty?...' -> complexity=0.183
'How do plants make their food?...' -> complexity=0.191
'What makes ice float on water?...' -> complexity=0.172
'Why do we have different seasons?...' -> complexity=0.157
'How do computers process information?...' -> complexity=0.176
'What is electricity and how it works?...' -> complexity=0.195
'Why do people dream during sleep?...' -> complexity=0.185

📊 Processing complex tasks...
'Explain the relationship between quantum mechanics...' -> complexity=0.210
'Analyze the economic impact of artificial intellig...' -> complexity=0.224
'What would happen if gravity suddenly became twice...' -> complexity=0.183
'Discuss the ethical implications of genetic engine...' -> complexity=0.206
'How might climate change affect global food securi...' -> complexity=0.213
'Evaluate the societal effects of social media on d...' -> complexity=0.201
'Compare the advantages and disadvantages of renewa...' -> complexity=0.204
'How do cultural differences affect international b...' -> complexity=0.191
'What are the long-term consequences of space explo...' -> complexity=0.200
'Analyze the philosophical implications of consciou...' -> complexity=0.232

============================================================
📈 ENHANCED MULTI-DIMENSIONAL RESULTS ANALYSIS
============================================================

1. Descriptive Statistics:
                 count   mean    std    min    max
true_complexity
complex             10  0.206  0.015  0.183  0.232
medium              10  0.181  0.014  0.157  0.207
simple              10  0.194  0.020  0.163  0.228
```




```
2. Statistical Significance Tests:
ANOVA F-statistic: 5.8874, p-value: 0.0076
✅ Significant difference between groups (p < 0.05)
Simple vs Complex tasks t-test: t=-1.569, p=0.1341

3. Correlation Analysis:
Correlation between complexity score and true complexity: 0.2678
❌ Weak correlation - Need to reconsider methodology

4. Routing Decision Analysis:
                 Total  Routed to Cloud  Routing Rate (%)
true_complexity
complex             10                0               0.0
```

```
complex           10                0               0.0
medium            10                0               0.0
simple            10                0               0.0

Routing Accuracy:
Simple task correct routing rate: 100.0%
Complex task correct routing rate: 0.0%

5. Feature Breakdown Analysis:
Average feature scores by complexity:
            entropy_score  variance_score  concentration_score  \
complexity
complex             0.093           0.237                0.821
medium              0.031           0.349                0.796
simple              0.022           0.416                0.791

            consistency_score
complexity
complex                 0.853
medium                  0.722
simple                  0.772

💾 Results saved to enhanced_results_fixed.csv

🎉 Stage 1 experiment completed!
```

```
"""
Fixed Cognitive Complexity-Based Attention Entropy Experiment
完全修复缩进错误的认知复杂度实验代码
"""
import torch
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy import stats
from transformers import AutoTokenizer, AutoModelForCausalLM
from typing import Dict, List


class EnhancedAttentionAnalyzer:
    def __init__(self):
        self.feature_weights = {
            'entropy': 0.8,
            'variance': 0.0,
            'concentration': 0.1,
            'cross_layer': 0.1
        }
        self.routing_threshold = 0.19
    
    def extract_multi_dimensional_features(self, text, model, tokenizer):
        inputs = tokenizer(text, return_tensors="pt", padding=True, truncation=True, max_length=512)
        
        with torch.no_grad():
            outputs = model(**inputs, output_attentions=True)
        
        all_attentions = outputs.attentions
        features = {}
        
        features.update(self.calculate_entropy_features(all_attentions, inputs))
        features.update(self.calculate_variance_features(all_attentions, inputs))
        features.update(self.calculate_concentration_features(all_attentions, inputs))
        features.update(self.calculate_cross_layer_features(all_attentions, inputs))
        
        return features
    
    def calculate_entropy_features(self, all_attentions, inputs):
        seq_len = inputs['attention_mask'].sum().item()
        last_layer = all_attentions[-1][0]
        
```

```
        last_token_entropy = self._calculate_token_entropy(last_layer[:, seq_len-1, :])
        
        all_token_entropies = []
        for i in range(seq_len):
            token_entropy = self._calculate_token_entropy(last_layer[:, i, :])
            all_token_entropies.append(token_entropy)
        avg_token_entropy = np.mean(all_token_entropies)
        
        key_token_entropy = self._calculate_key_token_entropy(last_layer, inputs, seq_len)
        
        return {
            'last_token_entropy': last_token_entropy,
            'avg_token_entropy': avg_token_entropy,
            'key_token_entropy': key_token_entropy,
            'entropy_variance': np.var(all_token_entropies) if len(all_token_entropies) > 1 else 0
        }
    
    def calculate_variance_features(self, all_attentions, inputs):
        seq_len = inputs['attention_mask'].sum().item()
        last_layer = all_attentions[-1][0]
        
        head_variances = []
        for head in range(last_layer.shape[0]):
            attn_weights = last_layer[head, seq_len-1, :seq_len]
            variance = torch.var(attn_weights).item()
            head_variances.append(variance)
        
        return {
            'avg_attention_variance': np.mean(head_variances),
            'max_attention_variance': np.max(head_variances),
            'variance_std': np.std(head_variances)
        }
    
    def calculate_concentration_features(self, all_attentions, inputs):
        seq_len = inputs['attention_mask'].sum().item()
        last_layer = all_attentions[-1][0]
        
        max_attentions = []
        gini_coefficients = []
        
        for head in range(last_layer.shape[0]):
            attn_weights = last_layer[head, seq_len-1, :seq_len]
            max_attention = torch.max(attn_weights).item()
            max_attentions.append(max_attention)
            gini = self._calculate_gini_coefficient(attn_weights)
            gini_coefficients.append(gini)
        
        return {
            'avg_max_attention': np.mean(max_attentions),
            'avg_gini_coefficient': np.mean(gini_coefficients),
        }
    
    def calculate_cross_layer_features(self, all_attentions, inputs):
        seq_len = inputs['attention_mask'].sum().item()
        
        if len(all_attentions) >= 3:
            last_layers = all_attentions[-3:]
            correlations = []
            
            for i in range(len(last_layers)-1):
                layer1 = last_layers[i][0][:, seq_len-1, :seq_len]
                layer2 = last_layers[i+1][0][:, seq_len-1, :seq_len]
                
                for head in range(layer1.shape[0]):
                    try:
                        head1_data = layer1[head].float()
                        head2_data = layer2[head].float()
                        
                        if torch.std(head1_data) > 1e-6 and torch.std(head2_data) > 1e-6:
                            corr_matrix = torch.corrcoef(torch.stack([head1_data, head2_data]))
                            corr = corr_matrix[0, 1]
                            if not torch.isnan(corr) and not torch.isinf(corr):
                                correlations.append(corr.item())
                    except:
                        continue
            
            cross_layer_consistency = np.mean(correlations) if correlations else 0
        else:
```

```
            cross_layer_consistency = 0

        return {
            'cross_layer_consistency': abs(cross_layer_consistency)
        }

    def _calculate_token_entropy(self, attention_weights):
        entropies = []
        for head_attn in attention_weights:
            head_attn = head_attn + 1e-9
            entropy = -torch.sum(head_attn * torch.log(head_attn))
            entropies.append(entropy.item())
        return np.mean(entropies)

    def _calculate_key_token_entropy(self, attention_matrix, inputs, seq_len):
        if seq_len > 3:
            middle_start = 1
            middle_end = seq_len - 1
            middle_entropies = []

            for i in range(middle_start, middle_end):
                token_entropy = self._calculate_token_entropy(attention_matrix[:, i, :])
                middle_entropies.append(token_entropy)

            return np.mean(middle_entropies) if middle_entropies else 0
        else:
            return self._calculate_token_entropy(attention_matrix[:, seq_len-1, :])

    def _calculate_gini_coefficient(self, weights):
        try:
            weights = weights.detach().cpu().numpy()
            weights = np.sort(weights)
            n = len(weights)
            if n == 0 or weights.sum() == 0:
                return 0

            cumsum = np.cumsum(weights)
            gini = (n + 1 - 2 * np.sum(cumsum) / cumsum[-1]) / n
            return max(0, min(1, gini))
        except:
            return 0

    def predict_complexity_enhanced(self, features):
        entropy_score = self._robust_normalize(features['avg_token_entropy'], 0.4, 1.2)
        variance_score = self._robust_normalize(features['avg_attention_variance'], 0, 0.2)
        concentration_score = self._robust_normalize(features['avg_max_attention'], 0.2, 0.8)
        consistency_score = self._robust_normalize(features['cross_layer_consistency'], 0, 1)

        complexity_score = (
            self.feature_weights['entropy'] * entropy_score +
            self.feature_weights['variance'] * variance_score +
            self.feature_weights['concentration'] * (1 - concentration_score) +
            self.feature_weights['cross_layer'] * consistency_score
        )

        complexity_score = max(0, min(1, complexity_score))

        return {
            'complexity_score': complexity_score,
            'features': features,
            'is_complex': complexity_score > self.routing_threshold,
            'avg_entropy': features['avg_token_entropy'],
            'head_entropies': [features['avg_token_entropy']] * 8,
            'feature_breakdown': {
                'entropy_score': entropy_score,
                'variance_score': variance_score,
                'concentration_score': concentration_score,
                'consistency_score': consistency_score
            }
        }

    def _robust_normalize(self, value, min_val, max_val, target_min=0, target_max=1):
        if max_val <= min_val:
            return target_min

        normalized = (value - min_val) / (max_val - min_val)
        normalized = max(0, min(1, normalized))
```

```
        
        return target_min + normalized * (target_max - target_min)
    
    def predict_complexity(self, attention_weights):
        entropies = []
        for head_attn in attention_weights:
            head_attn = head_attn + 1e-9
            entropy = -torch.sum(head_attn * torch.log(head_attn))
            entropies.append(entropy.item())
        
        avg_entropy = np.mean(entropies)
        complexity_score = self._robust_normalize(avg_entropy, 0.4, 1.2)
        
        return {
            'complexity_score': complexity_score,
            'avg_entropy': avg_entropy,
            'head_entropies': entropies,
            'is_complex': complexity_score > self.routing_threshold
        }


class CognitiveComplexityExperiment:
    def __init__(self, model_name="microsoft/DialoGPT-small", use_enhanced=True):
        self.tokenizer = AutoTokenizer.from_pretrained(model_name)
        self.model = AutoModelForCausalLM.from_pretrained(
            model_name, 
            output_attentions=True,
            attn_implementation="eager"
        )
        self.analyzer = EnhancedAttentionAnalyzer()
        self.use_enhanced = use_enhanced
        
        if self.tokenizer.pad_token is None:
            self.tokenizer.pad_token = self.tokenizer.eos_token
        
        self.task_dataset = {
            "simple": [
                "2 + 3 =",
                "Complete: The sky is ___",
                "True or False: Dogs are animals",
                "What comes after 5?",
                "Fill in: A, B, C, ___",
                "Yes or No: Is water wet?",
                "Complete: 1, 2, 3, ___",
                "What color is snow?",
                "How many legs does a dog have?",
                "Finish: Hello, my name ___"
            ],
            
            "medium": [
                "If it rains, what should you bring?",
                "Why do we wear coats in winter?",
                "What happens to ice when it melts?",
                "If you're hungry, what do you need?",
                "What tool cuts paper?",
                "Why do plants need sunlight?",
                "What makes things fall down?",
                "If it's dark, how do we see?",
                "Why do we sleep at night?",
                "What happens when you exercise?"
            ],
            
            "complex": [
                "Compare online vs traditional education effectiveness",
                "How would doubling gravity affect transportation?",
                "Analyze social media's impact on democracy",
                "Evaluate privacy vs security trade-offs in airports",
                "How might AI change future employment patterns?",
                "Discuss cultural factors in business negotiations",
                "What are the long-term effects of remote work?",
                "How do emotions influence investment decisions?",
                "Analyze the relationship between income and happiness",
                "Evaluate renewable energy policy trade-offs"
            ]
        }

    def extract_attention_features(self, text, method="last_token"):
        inputs = self.tokenizer(text, return_tensors="pt", padding=True, truncation=True, max_length=512)
```

```
        inputs = self.tokenizer(text, return_tensors="pt", padding=True, truncation=True, max_length=512)

        with torch.no_grad():
            outputs = self.model(**inputs, output_attentions=True)

        last_attention = outputs.attentions[-1][0]

        if method == "last_token":
            seq_len = inputs['attention_mask'].sum().item()
            last_token_attn = last_attention[:, seq_len-1, :]
            return last_token_attn
        elif method == "average":
            return last_attention
        else:
            raise ValueError(f"Unknown method: {method}")

    def run_single_task(self, text, complexity_label):
        if self.use_enhanced:
            features = self.analyzer.extract_multi_dimensional_features(text, self.model, self.tokenizer)
            result = self.analyzer.predict_complexity_enhanced(features)
        else:
            attention_weights = self.extract_attention_features(text)
            result = self.analyzer.predict_complexity(attention_weights)

        result['true_complexity'] = complexity_label
        result['task'] = text

        return result

    def run_comprehensive_experiment(self):
        method_name = "Enhanced Cognitive-Based" if self.use_enhanced else "Basic Entropy"
        print(f"🧪 Running {method_name} Classification Experiment...")
        print(f"📋 Using cognitive complexity-based task classification")
        print(f"🎯 Routing threshold: {self.analyzer.routing_threshold}")

        all_results = []

        for complexity, tasks in self.task_dataset.items():
            print(f"\n📊 Processing {complexity} tasks...")

            for task in tasks:
                result = self.run_single_task(task, complexity)
                all_results.append(result)
                routing_status = "→☁️" if result['is_complex'] else "→💻"
                print(f"'{task[:45]}...' -> {result['complexity_score']:.3f} {routing_status}")

        df = pd.DataFrame(all_results)
        return self.analyze_results(df)

    def analyze_results(self, df):
        method_name = "Enhanced Cognitive-Based" if self.use_enhanced else "Basic Entropy"
        print("\n" + "="*70)
        print(f"📈 {method_name.upper()} RESULTS ANALYSIS")
        print("="*70)

        summary = df.groupby('true_complexity')['complexity_score'].agg([
            'count', 'mean', 'std', 'min', 'max'
        ]).round(3)
        print("\n1. Descriptive Statistics:")
        print(summary)

        complex_mean = summary.loc['complex', 'mean']
        medium_mean = summary.loc['medium', 'mean']
        simple_mean = summary.loc['simple', 'mean']

        print(f"\n📊 Complexity Pattern Check:")
        print(f"Complex: {complex_mean:.3f} | Medium: {medium_mean:.3f} | Simple: {simple_mean:.3f}")

        if complex_mean > medium_mean > simple_mean:
            print("✅ Perfect ascending pattern: Simple < Medium < Complex")
        elif complex_mean > simple_mean:
            print("⚠️ Partial pattern: Complex > Simple, but Medium positioning needs work")
        else:
            print("❌ Pattern needs improvement")

        self.plot_results(df)
        self.statistical_tests(df)
        self.correlation_analysis(df)
```

```
        self.routing_analysis(df)

        if self.use_enhanced and 'feature_breakdown' in df.iloc[0]:
            self.feature_breakdown_analysis(df)

        return df

    def plot_results(self, df):
        plt.figure(figsize=(18, 6))

        plt.subplot(1, 4, 1)
        df.boxplot(column='complexity_score', by='true_complexity', ax=plt.gca())
        plt.title('Complexity Score Distribution')
        plt.ylabel('Complexity Score')
        plt.xlabel('True Complexity')

        plt.subplot(1, 4, 2)
        df.boxplot(column='avg_entropy', by='true_complexity', ax=plt.gca())
        plt.title('Average Attention Entropy Distribution')
        plt.ylabel('Average Entropy')
        plt.xlabel('True Complexity')

        plt.subplot(1, 4, 3)
        complexity_mapping = {'simple': 1, 'medium': 2, 'complex': 3}
        df['complexity_numeric'] = df['true_complexity'].map(complexity_mapping)
        plt.scatter(df['complexity_numeric'], df['complexity_score'], alpha=0.7)
        plt.xlabel('True Complexity')
        plt.ylabel('Predicted Complexity Score')
        plt.title('True vs Predicted Complexity')

        plt.subplot(1, 4, 4)
        routing_counts = df.groupby(['true_complexity', 'is_complex']).size().unstack(fill_value=0)
        routing_counts.plot(kind='bar', ax=plt.gca(), color=['lightblue', 'orange'])
        plt.title('Routing Decisions by Complexity')
        plt.ylabel('Number of Tasks')
        plt.xlabel('True Complexity')
        plt.legend(['Local (SLM)', 'Cloud (LLM)'])
        plt.xticks(rotation=0)

        plt.tight_layout()
        plt.show()

    def statistical_tests(self, df):
        print("\n2. Statistical Significance Tests:")

        simple_scores = df[df['true_complexity'] == 'simple']['complexity_score']
        medium_scores = df[df['true_complexity'] == 'medium']['complexity_score']
        complex_scores = df[df['true_complexity'] == 'complex']['complexity_score']

        f_stat, p_value = stats.f_oneway(simple_scores, medium_scores, complex_scores)
        print(f"ANOVA F-statistic: {f_stat:.4f}, p-value: {p_value:.4f}")

        if p_value < 0.001:
            print("✅ Highly significant difference between groups (p < 0.001)")
        elif p_value < 0.01:
            print("✅ Very significant difference between groups (p < 0.01)")
        elif p_value < 0.05:
            print("✅ Significant difference between groups (p < 0.05)")
        else:
            print("❌ No significant difference between groups (p >= 0.05)")

        from scipy.stats import ttest_ind
        t1, p1 = ttest_ind(simple_scores, complex_scores)
        t2, p2 = ttest_ind(simple_scores, medium_scores)
        t3, p3 = ttest_ind(medium_scores, complex_scores)

        print(f"Pairwise t-tests:")
        print(f"  Simple vs Complex: t={t1:.3f}, p={p1:.4f}")
        print(f"  Simple vs Medium:  t={t2:.3f}, p={p2:.4f}")
        print(f"  Medium vs Complex: t={t3:.3f}, p={p3:.4f}")

    def correlation_analysis(self, df):
        print("\n3. Correlation Analysis:")

        complexity_mapping = {'simple': 1, 'medium': 2, 'complex': 3}
        df['complexity_numeric'] = df['true_complexity'].map(complexity_mapping)

        correlation = df['complexity_score'].corr(df['complexity_numeric'])
```

```
        correlation = df['complexity_score'].corr(df['complexity_numeric'])
        print(f"Correlation between complexity score and true complexity: {correlation:.4f}")
        
        if correlation > 0.7:
            print("✅ Very strong positive correlation - Excellent validation!")
        elif correlation > 0.5:
            print("✅ Strong positive correlation - Good validation!")
        elif correlation > 0.3:
            print("⚠️ Moderate correlation - Acceptable but can improve")
        else:
            print("❌ Weak correlation - Need methodology adjustment")
        
        r_squared = correlation ** 2
        print(f"R² (explained variance): {r_squared:.3f} ({r_squared*100:.1f}%)")

    def routing_analysis(self, df):
        print("\n4. Routing Decision Analysis:")
        print(f"Using threshold: {self.analyzer.routing_threshold}")
        
        routing_stats = df.groupby('true_complexity')['is_complex'].agg([
            'count', 'sum', lambda x: (x.sum() / len(x) * 100)
        ]).round(1)
        routing_stats.columns = ['Total', 'Routed to Cloud', 'Routing Rate (%)']
        print("\nRouting Statistics:")
        print(routing_stats)
        
        simple_correct = (df[df['true_complexity'] == 'simple']['is_complex'] == False).sum()
        complex_correct = (df[df['true_complexity'] == 'complex']['is_complex'] == True).sum()
        
        simple_total = len(df[df['true_complexity'] == 'simple'])
        complex_total = len(df[df['true_complexity'] == 'complex'])
        
        print(f"\nRouting Accuracy:")
        print(f"Simple tasks correctly kept local: {simple_correct/simple_total*100:.1f}%")
        print(f"Complex tasks correctly sent to cloud: {complex_correct/complex_total*100:.1f}%")
        
        total_correct = simple_correct + complex_correct
        total_decisive = simple_total + complex_total
        overall_accuracy = total_correct / total_decisive * 100
        print(f"Overall routing accuracy: {overall_accuracy:.1f}%")

    def feature_breakdown_analysis(self, df):
        print("\n5. Feature Breakdown Analysis:")
        
        if 'feature_breakdown' not in df.iloc[0]:
            print("Feature breakdown not available")
            return
        
        feature_data = []
        for _, row in df.iterrows():
            breakdown = row['feature_breakdown']
            feature_data.append({
                'complexity': row['true_complexity'],
                'entropy_score': breakdown['entropy_score'],
                'variance_score': breakdown['variance_score'],
                'concentration_score': breakdown['concentration_score'],
                'consistency_score': breakdown['consistency_score']
            })
        
        feature_df = pd.DataFrame(feature_data)
        feature_summary = feature_df.groupby('complexity')[
            ['entropy_score', 'variance_score', 'concentration_score', 'consistency_score']
        ].mean().round(3)
        
        print("Average feature scores by complexity:")
        print(feature_summary)

    def save_results(self, df, filename="cognitive_complexity_results.csv"):
        df.to_csv(filename, index=False)
        print(f"\n💾 Results saved to {filename}")

    def generate_report(self, df):
        print("\n" + "="*70)
        print("📋 EXPERIMENT SUMMARY REPORT")
        print("="*70)
        
        total_tasks = len(df)
        correlation = df['complexity_score'].corr(
```

```
            df['true_complexity'].map({'simple': 1, 'medium': 2, 'complex': 3})
        )
        
        simple_scores = df[df['true_complexity'] == 'simple']['complexity_score']
        medium_scores = df[df['true_complexity'] == 'medium']['complexity_score']
        complex_scores = df[df['true_complexity'] == 'complex']['complexity_score']
        f_stat, p_value = stats.f_oneway(simple_scores, medium_scores, complex_scores)
        
        simple_correct = (df[df['true_complexity'] == 'simple']['is_complex'] == False).sum()
        complex_correct = (df[df['true_complexity'] == 'complex']['is_complex'] == True).sum()
        routing_accuracy = (simple_correct + complex_correct) / (len(simple_scores) + len(complex_scores)) * 100
        
        print(f"""
🎯 KEY FINDINGS:
├── Total tasks analyzed: {total_tasks}
├── Correlation coefficient: {correlation:.3f}
├── ANOVA significance: p={p_value:.4f}
├── Routing accuracy: {routing_accuracy:.1f}%
└── Method: {'Enhanced Multi-dimensional' if self.use_enhanced else 'Basic Entropy'}

📊 COMPLEXITY SCORES:
├── Simple:  {simple_scores.mean():.3f} ± {simple_scores.std():.3f}
├── Medium:  {medium_scores.mean():.3f} ± {medium_scores.std():.3f}
└── Complex: {complex_scores.mean():.3f} ± {complex_scores.std():.3f}

🎭 PERFORMANCE ASSESSMENT:
""")
        
        if correlation > 0.5 and p_value < 0.05 and routing_accuracy > 70:
            print("✅ EXCELLENT: Strong validation of attention entropy hypothesis!")
        elif correlation > 0.3 and p_value < 0.05:
            print("✅ GOOD: Solid evidence supporting the hypothesis with room for optimization")
        elif p_value < 0.05:
            print("⚠️ MODERATE: Statistical significance achieved but correlation needs improvement")
        else:
            print("❌ NEEDS WORK: Hypothesis validation requires methodology adjustment")


def quick_run_cognitive_experiment():
    experiment = CognitiveComplexityExperiment(use_enhanced=True)
    results_df = experiment.run_comprehensive_experiment()
    experiment.save_results(results_df, "cognitive_complexity_results.csv")
    experiment.generate_report(results_df)
    return results_df


if __name__ == "__main__":
    print("🎯 Running Enhanced Cognitive Complexity Experiment...")
    results = quick_run_cognitive_experiment()
    print("\n🎉 Experiment completed successfully!")
```