```python
"""
OL Causal Cascade - Enhanced Intelligence Framework
Combining NFL-specific modeling with cross-domain validation
"""

import pandas as pd
import numpy as np
from abc import ABC, abstractmethod
from typing import Dict, List, Tuple, Any, Optional
from dataclasses import dataclass
from enum import Enum
import logging

# Configure logging for validation tracking
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

class Domain(Enum):
    NFL = "nfl"
    NBA = "nba"
    HEALTHCARE = "healthcare"
    ECONOMICS = "economics"
    VALIDATION = "validation"

@dataclass
class ValidationDataset:
    name: str
    df: pd.DataFrame
    ground_truth_effect: float
    treatment_var: str
    outcome_var: str
    confounders: List[str]
    domain: Domain

@dataclass
class CausalValidationResult:
    dataset_name: str
    estimated_effect: float
    ground_truth: float
    absolute_error: float
    relative_error: float
```

```
    confidence_interval: Tuple[float, float]
    passed_validation: bool
    execution_time: float

class CausalIntelligenceEngine(ABC):
    """Abstract base class for domain-agnostic causal
inference"""

    def __init__(self, domain: Domain):
        self.domain = domain
        self.validation_results: List[CausalValidationResult] =
[]

    @abstractmethod
    def construct_dag(self, data: pd.DataFrame) -> Dict[str,
Any]:
        """Build domain-specific DAG structure"""
        pass

    @abstractmethod
    def estimate_treatment_effects(self, data: pd.DataFrame,
                                   treatment: str, outcome: str)
-> Dict[str, float]:
        """Estimate causal effects using domain-appropriate
methods"""
        pass

    @abstractmethod
    def identify_confounders(self, data: pd.DataFrame) ->
List[str]:
        """Domain-specific confounder identification"""
        pass

class CausalFrameworkValidator:
    """Validates causal inference methodology across known
datasets"""

    def __init__(self, engine: CausalIntelligenceEngine):
        self.engine = engine
        self.validation_threshold = 0.05  # 5% error tolerance
        self.results: List[CausalValidationResult] = []
```

```
    def create_synthetic_datasets(self) ->
List[ValidationDataset]:
        """Generate synthetic datasets with known causal
effects"""
        datasets = []

        # Synthetic Dataset 1: Simple Treatment Effect
        np.random.seed(42)
        n = 1000

        # Confounders
        X1 = np.random.normal(0, 1, n)
        X2 = np.random.normal(0, 1, n)

        # Treatment assignment (confounded)
        treatment_prob = 1 / (1 + np.exp(-(0.5 * X1 + 0.3 *
X2)))
        T = np.random.binomial(1, treatment_prob)

        # Outcome (true effect = 2.0)
        Y = 2.0 * T + 1.5 * X1 + 1.0 * X2 + np.random.normal(0,
1, n)

        synthetic_df = pd.DataFrame({
            'treatment': T,
            'outcome': Y,
            'confounder_1': X1,
            'confounder_2': X2
        })

        datasets.append(ValidationDataset(
            name="Synthetic_Simple_Treatment",
            df=synthetic_df,
            ground_truth_effect=2.0,
            treatment_var='treatment',
            outcome_var='outcome',
            confounders=['confounder_1', 'confounder_2'],
            domain=Domain.VALIDATION
        ))
```

```
        return datasets

    def validate_dag_approach(self, dataset: ValidationDataset)
-> CausalValidationResult:
        """Test causal inference on dataset with known ground
truth"""
        import time
        start_time = time.time()

        try:
            # Apply our causal inference method
            effects = self.engine.estimate_treatment_effects(
                dataset.df,
                dataset.treatment_var,
                dataset.outcome_var
            )

            estimated_effect = effects.get('ate', 0.0)  #
Average Treatment Effect

            # Calculate validation metrics
            abs_error = abs(dataset.ground_truth_effect -
estimated_effect)
            rel_error = abs_error /
abs(dataset.ground_truth_effect) if dataset.ground_truth_effect
!= 0 else float('inf')

            # Mock confidence interval (would come from
bootstrapping/inference)
            ci_lower = estimated_effect - 1.96 * 0.1  #
Placeholder std error
            ci_upper = estimated_effect + 1.96 * 0.1

            passed = abs_error < self.validation_threshold
            execution_time = time.time() - start_time

            result = CausalValidationResult(
                dataset_name=dataset.name,
                estimated_effect=estimated_effect,
                ground_truth=dataset.ground_truth_effect,
                absolute_error=abs_error,
```

```
                relative_error=rel_error,
                confidence_interval=(ci_lower, ci_upper),
                passed_validation=passed,
                execution_time=execution_time
            )

            logger.info(f"Validation Results for
{dataset.name}:")
            logger.info(f"  Estimated: {estimated_effect:.4f}")
            logger.info(f"  Ground Truth:
{dataset.ground_truth_effect:.4f}")
            logger.info(f"  Error: {abs_error:.4f} ({'PASS' if
passed else 'FAIL'})")

            return result

        except Exception as e:
            logger.error(f"Validation failed for {dataset.name}:
{str(e)}")
            return CausalValidationResult(
                dataset_name=dataset.name,
                estimated_effect=0.0,
                ground_truth=dataset.ground_truth_effect,
                absolute_error=float('inf'),
                relative_error=float('inf'),
                confidence_interval=(0.0, 0.0),
                passed_validation=False,
                execution_time=time.time() - start_time
            )

    def run_full_validation_suite(self) -> Dict[str, Any]:
        """Execute complete validation across all test
datasets"""

        # Get synthetic datasets
        test_datasets = self.create_synthetic_datasets()

        # Run validation on each dataset
        for dataset in test_datasets:
            result = self.validate_dag_approach(dataset)
            self.results.append(result)
```

```
        # Generate summary report
        passed_count = sum(1 for r in self.results if
r.passed_validation)
        total_count = len(self.results)

        avg_abs_error = np.mean([r.absolute_error for r in
self.results if r.absolute_error != float('inf')])
        avg_rel_error = np.mean([r.relative_error for r in
self.results if r.relative_error != float('inf')])

        summary = {
            'total_tests': total_count,
            'passed_tests': passed_count,
            'pass_rate': passed_count / total_count if
total_count > 0 else 0,
            'average_absolute_error': avg_abs_error,
            'average_relative_error': avg_rel_error,
            'methodology_validated': passed_count >= total_count
* 0.8  # 80% pass threshold
        }

        logger.info(f"\n=== VALIDATION SUMMARY ===")
        logger.info(f"Pass Rate: {summary['pass_rate']:.1%}
({passed_count}/{total_count})")
        logger.info(f"Avg Absolute Error: {avg_abs_error:.4f}")
        logger.info(f"Methodology Status: {'VALIDATED' if
summary['methodology_validated'] else 'NEEDS_IMPROVEMENT'}")

        return summary

class NFLCausalEngine(CausalIntelligenceEngine):
    """NFL-specific implementation of causal intelligence"""

    def __init__(self):
        super().__init__(Domain.NFL)
        self.lt_cascade_dag = None

    def construct_dag(self, data: pd.DataFrame) -> Dict[str,
Any]:
        """Build NFL LT Causal Cascade DAG"""
```

```
        # Primary Impact Layer
        primary_nodes = [
            'lt_injury_severity',
            'lt_replacement_quality',
            'immediate_protection_gap'
        ]

        # Secondary Impact Layer
        secondary_nodes = [
            'te_chip_frequency',
            'rb_max_protect_usage',
            'qb_time_to_throw',
            'route_tree_modification'
        ]

        # Tertiary Impact Layer
        tertiary_nodes = [
            'offensive_epa_change',
            'passing_success_rate',
            'rushing_efficiency',
            'red_zone_conversion'
        ]

        # Define causal relationships
        dag_structure = {
            'primary_layer': primary_nodes,
            'secondary_layer': secondary_nodes,
            'tertiary_layer': tertiary_nodes,
            'edges': [
                ('lt_injury_severity',
'immediate_protection_gap'),
                ('lt_replacement_quality',
'immediate_protection_gap'),
                ('immediate_protection_gap',
'te_chip_frequency'),
                ('immediate_protection_gap',
'qb_time_to_throw'),
                ('te_chip_frequency',
'route_tree_modification'),
                ('qb_time_to_throw', 'passing_success_rate'),
```

```
                ('route_tree_modification',
'offensive_epa_change')
            ]
        }

        self.lt_cascade_dag = dag_structure
        return dag_structure

    def estimate_treatment_effects(self, data: pd.DataFrame,
                                   treatment: str, outcome: str)
-> Dict[str, float]:
        """Estimate NFL-specific causal effects"""

        # Placeholder for actual causal inference
        # In practice, this would use DoWhy, CausalML, or custom
implementation

        # Mock estimation using simple regression for validation
        from sklearn.linear_model import LinearRegression

        # Identify confounders automatically
        confounders = self.identify_confounders(data)

        # Prepare features
        X_cols = [treatment] + confounders
        X = data[X_cols].fillna(0)
        y = data[outcome].fillna(0)

        # Simple regression estimate (would be replaced with
proper causal method)
        model = LinearRegression()
        model.fit(X, y)

        treatment_effect = model.coef_[0]  # Coefficient of
treatment variable

        return {
            'ate': treatment_effect,  # Average Treatment Effect
            'method': 'linear_regression_placeholder',
            'confounders_controlled': confounders
        }
```

```
    def identify_confounders(self, data: pd.DataFrame) ->
List[str]:
        """Identify NFL-specific confounders"""

        # Domain knowledge-based confounder identification
        potential_confounders = [
            'team_strength', 'opponent_strength',
'game_situation',
            'weather_conditions', 'home_away', 'week_number',
            'qb_experience', 'offensive_line_continuity'
        ]

        # Return confounders that exist in the data
        available_confounders = [col for col in
potential_confounders if col in data.columns]

        return available_confounders

# Example usage and testing framework
def run_methodology_validation():
    """Execute the full validation pipeline"""

    # Initialize NFL engine
    nfl_engine = NFLCausalEngine()

    # Create validator
    validator = CausalFrameworkValidator(nfl_engine)

    # Run validation suite
    validation_summary = validator.run_full_validation_suite()

    return validation_summary, validator.results

if __name__ == "__main__":
    # Run validation
    summary, detailed_results = run_methodology_validation()

    print("\n=== METHODOLOGY VALIDATION COMPLETE ===")
    print(f"Framework Ready for NFL Application:
{summary['methodology_validated']}")
```