```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
```

```python
    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] = []
```

```python
    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
        ))
```

```python
        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,
```

```python
                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)
```

```python
        # 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 with 
market integration"""

    def __init__(self):
        super().__init__(Domain.NFL)
        self.lt_cascade_dag = None
        self.kalman_filter = None  # Dynamic team strength model
        self.market_dag = None     # Market behavior modeling
```

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

        # Layer 0: Latent State (Dynamic Strength via Kalman Filter)
        latent_nodes = [
            'team_latent_strength',
            'opponent_latent_strength',
            'situational_context'
        ]

        # Primary Impact Layer (On-Field Causal Chain)
        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 (On-Field Outcomes)
        tertiary_nodes = [
            'offensive_epa_change',
            'passing_success_rate',
            'rushing_efficiency',
            'red_zone_conversion',
            'theoretical_point_impact'  # Our "true" game impact
        ]

        # Market Layer (Betting Market Dynamics)
        market_nodes = [
            'opening_line',
            'public_bet_percentage',
```

```
            'sharp_money_indicators',
            'line_movement_velocity',
            'closing_line_value',
            'final_market_spread'
        ]

        # Value Detection Layer
        value_nodes = [
            'model_predicted_spread',
            'market_predicted_spread',
            'value_gap',  # The money-making differential
            'bet_recommendation'
        ]

        # Enhanced causal relationships
        dag_structure = {
            'latent_layer': latent_nodes,
            'primary_layer': primary_nodes,
            'secondary_layer': secondary_nodes,
            'tertiary_layer': tertiary_nodes,
            'market_layer': market_nodes,
            'value_layer': value_nodes,
            'edges': [
                # Core LT injury cascade
                ('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'),

                # Latent state influences
                ('team_latent_strength',
'offensive_epa_change'),
```

```
                ('opponent_latent_strength', 'offensive_epa_change'),
                ('situational_context', 'offensive_epa_change'),

                # On-field to theoretical impact
                ('offensive_epa_change', 'theoretical_point_impact'),
                ('passing_success_rate', 'theoretical_point_impact'),
                ('rushing_efficiency', 'theoretical_point_impact'),

                # Market dynamics
                ('theoretical_point_impact', 'opening_line'),
                ('public_bet_percentage', 'line_movement_velocity'),
                ('sharp_money_indicators', 'line_movement_velocity'),
                ('line_movement_velocity', 'final_market_spread'),

                # Value gap calculation
                ('theoretical_point_impact', 'model_predicted_spread'),
                ('final_market_spread', 'market_predicted_spread'),
                ('model_predicted_spread', 'value_gap'),
                ('market_predicted_spread', 'value_gap'),
                ('value_gap', 'bet_recommendation')
            ]
        }

        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 causal effects with value gap calculation"""
```

```python
        # Step 1: Update latent team strengths using Kalman filter
        team_strengths = self._update_latent_strengths(data)

        # Step 2: Estimate pure on-field causal effect
        on_field_effect = self._estimate_on_field_impact(data, treatment, outcome, team_strengths)

        # Step 3: Convert to theoretical point spread
        theoretical_spread = self._convert_to_point_impact(on_field_effect)

        # Step 4: Calculate market-adjusted value gap
        market_spread = data.get('final_market_spread', 0.0).iloc[-1] if len(data) > 0 else 0.0
        value_gap = theoretical_spread - market_spread

        # Step 5: Generate bet recommendation
        bet_recommendation = self._generate_bet_signal(value_gap)

        return {
            'ate': on_field_effect,  # Average Treatment Effect (on-field)
            'theoretical_spread': theoretical_spread,
            'market_spread': market_spread,
            'value_gap': value_gap,
            'bet_recommendation': bet_recommendation,
            'method': 'enhanced_causal_with_market_integration',
            'confounders_controlled': self.identify_confounders(data)
        }

    def _update_latent_strengths(self, data: pd.DataFrame) -> Dict[str, float]:
        """Update dynamic team strengths using Kalman filter approach"""
        # Placeholder for Kalman filter implementation
        # In practice, this would maintain running estimates of team quality
```

```python
        if self.kalman_filter is None:
            # Initialize with simple averages, upgrade to proper Kalman filter
            team_strength = data.get('recent_offensive_epa', [0.0]).mean()
            opp_strength = data.get('recent_defensive_epa', [0.0]).mean()
        else:
            # Use Kalman filter for dynamic estimation
            team_strength = self.kalman_filter.update_team_strength(data)
            opp_strength = self.kalman_filter.update_opponent_strength(data)

        return {
            'team_latent_strength': team_strength,
            'opponent_latent_strength': opp_strength
        }

    def _estimate_on_field_impact(self, data: pd.DataFrame, treatment: str,
                                outcome: str, team_strengths: Dict[str, float]) -> float:
        """Estimate pure on-field causal effect controlling for latent strengths"""
        from sklearn.linear_model import LinearRegression

        # Enhanced confounders including dynamic team strength
        confounders = self.identify_confounders(data)

        # Add latent strengths as confounders
        enhanced_data = data.copy()
        for strength_var, strength_val in team_strengths.items():
            enhanced_data[strength_var] = strength_val
            confounders.append(strength_var)

        # Causal estimation with enhanced controls
        X_cols = [treatment] + confounders
        X = enhanced_data[X_cols].fillna(0)
        y = enhanced_data[outcome].fillna(0)
```

```python
        model = LinearRegression()
        model.fit(X, y)

        return model.coef_[0]  # Treatment effect coefficient

    def _convert_to_point_impact(self, epa_effect: float) -> 
float:
        """Convert EPA impact to point spread equivalent"""
        # EPA to points conversion (roughly 1 EPA ≈ 7 points)
        # This would be calibrated using historical data
        points_per_epa = 7.0
        return epa_effect * points_per_epa

    def _generate_bet_signal(self, value_gap: float) -> str:
        """Generate betting recommendation based on value gap"""
        if abs(value_gap) < 1.0:
            return "NO_BET"  # Insufficient edge
        elif value_gap > 1.0:
            return "BET_UNDER"  # Market overreacting to injury
        else:
            return "BET_OVER"   # Market underreacting to injury

    def identify_confounders(self, data: pd.DataFrame) -> 
List[str]:
        """Identify enhanced NFL confounders including market 
variables"""

        # On-field confounders (traditional)
        on_field_confounders = [
            'qb_experience', 'offensive_line_continuity', 
'weather_conditions',
            'home_away', 'week_number', 'game_situation', 
'injury_report_status'
        ]

        # Market confounders (from Deciphering Framework)
        market_confounders = [
            'public_bet_percentage', 'sharp_money_indicators', 
'line_movement_velocity',
```

```python
            'opening_line', 'injury_announcement_timing',
'media_coverage_volume'
        ]

        # Dynamic state confounders (Kalman filter outputs)
        dynamic_confounders = [
            'team_latent_strength', 'opponent_latent_strength',
            'recent_offensive_trend', 'recent_defensive_trend'
        ]

        # Combine all potential confounders
        all_potential_confounders = on_field_confounders +
market_confounders + dynamic_confounders

        # Return confounders that exist in the data
        available_confounders = [col for col in
all_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']}")
```