```
# racecard/services/scoring_service.py
import os
import sys
sys.path.append(os.path.join(os.path.dirname(__file__), '..'))

from datetime import date, timedelta
from django.db.models import Avg, Count, Q
from racecard.models import Horse, Run, Race, HorseScore
from services.class_analysis import ClassAnalysisService

class HorseScoringService:

    def __init__(self, horse, race):
        self.horse = horse
        self.race = race
        self.runs = Run.objects.filter(horse=horse).order_by('-run_date')

    def calculate_merit_score(self):
        """Score based on horse's merit rating"""
        base_merit = self.horse.horse_merit or 0
        return min(base_merit / 100, 1.0)  # Normalize to 0-1

    def calculate_form_score(self):
        """Score based on recent form"""
        recent_runs = self.runs[:6]  # Last 6 runs
        if not recent_runs:
            return 0.5  # Neutral score for no data

        positions = []
        for run in recent_runs:
            try:
                pos = float(run.position) if run.position and
run.position.isdigit() else None
                if pos:
                    positions.append(pos)
            except:
                continue

        if not positions:
            return 0.5

        # Weight recent runs more heavily
        weighted_avg = sum(pos * (0.8 ** i) for i, pos in enumerate(positions))
        total_weight = sum(0.8 ** i for i in range(len(positions)))
        avg_position = weighted_avg / total_weight

        return max(0, 1 - (avg_position / 12))  # Better positions score higher

    def calculate_class_score(self):
        """Score based on class suitability"""
        class_weights = {
            'G1': 1.0, 'G2': 0.9, 'G3': 0.8, 'L': 0.7,
            'Cl1': 0.6, 'Cl2': 0.5, 'Cl3': 0.4, 'Cl4': 0.3,
            'Md': 0.2
        }

        current_class = self.race.race_class or ''
        for class_key, weight in class_weights.items():
            if class_key.lower() in current_class.lower():
```

```
                return weight

        return 0.5  # Default

    def calculate_distance_score(self):
        """Score based on distance suitability"""
        target_distance = self._parse_distance(self.race.race_distance)
        if not target_distance:
            return 0.5

        distance_performances = []
        for run in self.runs[:10]:
            run_distance = self._parse_distance(run.distance)
            if run_distance and run.position and run.position.isdigit():
                try:
                    position = float(run.position)
                    # Score based on performance at similar distances
                    distance_diff = abs(run_distance - target_distance)
                    similarity = max(0, 1 - (distance_diff / 400))  # 400m
tolerance
                    performance = max(0, 1 - (position / 12))  # Better positions
score higher
                    distance_performances.append(performance * similarity)
                except:
                    continue

        return sum(distance_performances) / len(distance_performances) if
distance_performances else 0.5

    def calculate_consistency_score(self):
        """Score based on performance consistency"""
        recent_positions = []
        for run in self.runs[:8]:
            if run.position and run.position.isdigit():
                try:
                    recent_positions.append(float(run.position))
                except:
                    continue

        if len(recent_positions) < 3:
            return 0.5

        avg_position = sum(recent_positions) / len(recent_positions)
        variance = sum((pos - avg_position) ** 2 for pos in recent_positions) /
len(recent_positions)
        consistency = max(0, 1 - (variance / 20))  # Lower variance = higher
consistency

        return consistency

    def _parse_distance(self, distance_str):
        """Parse distance string to meters"""
        if not distance_str:
            return None
        try:
            return int(''.join(filter(str.isdigit, str(distance_str))))
        except:
            return None
```

```
    def calculate_composite_score(self):
        """Calculate overall composite score"""
        weights = {
            'merit': 0.25,
            'form': 0.30,
            'class': 0.15,
            'distance': 0.20,
            'consistency': 0.10
        }

        scores = {
            'merit': self.calculate_merit_score(),
            'form': self.calculate_form_score(),
            'class': self.calculate_class_score(),
            'distance': self.calculate_distance_score(),
            'consistency': self.calculate_consistency_score()
        }

        composite = sum(scores[factor] * weights[factor] for factor in weights)
        return min(max(composite, 0), 1.0)  # Ensure between 0-1

    # racecard/services/scoring_service.py - Update the create_score_record method
    def create_score_record(self):
        """Create or update a comprehensive score record"""
        composite_score = self.calculate_composite_score()

        # Use update_or_create to handle existing records
        score_record, created = HorseScore.objects.update_or_create(
            horse=self.horse,
            race=self.race,
            defaults={
                'overall_score': composite_score,
                'merit_score': self.calculate_merit_score(),
                'form_score': self.calculate_form_score(),
                'class_score': self.calculate_class_score(),
                'distance_score': self.calculate_distance_score(),
                'consistency_score': self.calculate_consistency_score()
            }
        )

        return score_record, created



    def __init__(self, horse, race):
        self.horse = horse
        self.race = race
        self.runs = Run.objects.filter(horse=horse).order_by('-run_date')
        self.class_analyzer = ClassAnalysisService()

    def calculate_class_score(self):
        """Enhanced class suitability score"""
        return self.class_analyzer.calculate_class_suitability(self.horse,
self.race)

    def calculate_composite_score(self):
        """Calculate overall composite score with enhanced class analysis"""
        weights = {
            'merit': 0.20,
```

```
            'form': 0.25,
            'class': 0.25,  # Increased weight for class analysis
            'distance': 0.15,
            'consistency': 0.10,
            'class_trend': 0.05  # New: class movement trend
        }

        scores = {
            'merit': self.calculate_merit_score(),
            'form': self.calculate_form_score(),
            'class': self.calculate_class_score() / 100,  # Convert to 0-1
            'distance': self.calculate_distance_score(),
            'consistency': self.calculate_consistency_score(),
            'class_trend': self._calculate_class_trend_score()
        }

        composite = sum(scores[factor] * weights[factor] for factor in weights)
        return min(max(composite, 0), 1.0)

    def _calculate_class_trend_score(self):
        """Score based on class movement trend"""
        trend = self.class_analyzer.get_class_trend(self.horse)

        if trend == "moving_up":
            return 0.8  # Slight penalty for moving up in class
        elif trend == "moving_down":
            return 1.2  # Bonus for moving down in class
        else:
            return 1.0  # Neutral for stable
```