```
# racecard/services/class_analysis.py
import json
import os
import re
from django.conf import settings

class ClassAnalysisService:

    def __init__(self):
        self.class_weights = self._load_class_weights()

    def _load_class_weights(self):
        """Load class weights from JSON file"""
        weights_path = os.path.join(settings.BASE_DIR, 'racecard', 'data',
'class_weights.json')
        try:
            with open(weights_path, 'r') as f:
                data = json.load(f)
                return {cls['abbreviation']: cls for cls in data['classes']}
        except FileNotFoundError:
            # Fallback to default weights
            return self._get_default_weights()
        except json.JSONDecodeError:
            # Fallback to default weights if JSON is invalid
            return self._get_default_weights()

    def _get_default_weights(self):
        """Default weights if JSON file not found"""
        return {
            'MP': {'name': 'Maiden Plate', 'weight': 1},
            'MP-F': {'name': 'Maiden Plate (Fillies)', 'weight': 2},
            'OM': {'name': 'Open Maiden', 'weight': 3},
            'Juv': {'name': 'Juvenile', 'weight': 4},
            'Cl6': {'name': 'Class 6', 'weight': 5},
            'Cl5': {'name': 'Class 5', 'weight': 6},
            'Cl4': {'name': 'Class 4', 'weight': 7},
            'Cl3': {'name': 'Class 3', 'weight': 8},
            'Cl2': {'name': 'Class 2', 'weight': 9},
            'Cl1': {'name': 'Class 1', 'weight': 10},
            'L': {'name': 'Listed', 'weight': 11},
            'G3': {'name': 'Group 3', 'weight': 12},
            'G2': {'name': 'Group 2', 'weight': 13},
            'G1': {'name': 'Group 1', 'weight': 14},
            'Hcp': {'name': 'Handicap', 'weight': 15},
            'MR': {'name': 'Merit Rated', 'weight': 16},
            'BM': {'name': 'Benchmark', 'weight': 17},
            'Stk': {'name': 'Stakes', 'weight': 18},
            'Cond': {'name': 'Conditions', 'weight': 19},
            'Allow': {'name': 'Allowance', 'weight': 20},
            'App': {'name': 'Apprentice', 'weight': 21},
            'Nov': {'name': 'Novice', 'weight': 22},
            'Grad': {'name': 'Graduation', 'weight': 23},
            'Rest': {'name': 'Restricted', 'weight': 24}
        }

    def get_class_weight(self, race_class):
        """Get weight for a given race class string"""
        if not race_class:
            return 0
```

```
    race_class = race_class.upper().strip()

    # Try to match abbreviations first
    for abbrev, cls_data in self.class_weights.items():
        if abbrev.upper() in race_class:
            return cls_data['weight']

    # Try to match by name
    for cls_data in self.class_weights.values():
        if cls_data['name'].upper() in race_class:
            return cls_data['weight']

    # Default based on merit rating if present
    merit_match = re.search(r'MR\s*(\d+)', race_class)
    if merit_match:
        return int(merit_match.group(1)) / 2  # Scale merit rating to weight

    # Default for unknown classes
    return 25

def analyze_horse_class_history(self, horse):
    """Analyze a horse's class history from last 4 runs"""
    # Use absolute import instead of relative import
    from racecard.models import Run

    runs = Run.objects.filter(horse=horse).order_by('-run_date')[:4]

    if not runs:
        return {
            'average_class_weight': 0,
            'class_consistency': 0,
            'highest_class': 0,
            'lowest_class': 0,
            'runs_analyzed': 0
        }

    class_weights = []
    for run in runs:
        if run.race_class:
            weight = self.get_class_weight(run.race_class)
            class_weights.append(weight)

    if not class_weights:
        return {
            'average_class_weight': 0,
            'class_consistency': 0,
            'highest_class': 0,
            'lowest_class': 0,
            'runs_analyzed': 0
        }

    # Calculate metrics
    avg_weight = sum(class_weights) / len(class_weights)
    max_weight = max(class_weights)
    min_weight = min(class_weights)

    # Consistency (lower std dev = more consistent)
    variance = sum((w - avg_weight) ** 2 for w in class_weights) /
```

```
len(class_weights)
        consistency = max(0, 100 - (variance * 10))  # Convert to 0-100 scale

        return {
            'average_class_weight': avg_weight,
            'class_consistency': consistency,
            'highest_class': max_weight,
            'lowest_class': min_weight,
            'runs_analyzed': len(class_weights),
            'recent_classes': class_weights
        }

    def calculate_class_suitability(self, horse, current_race):
        """Calculate how suitable the horse is for the current race class"""
        # Get current race class weight
        current_class_weight = self.get_class_weight(current_race.race_class)

        # Analyze horse's class history
        class_history = self.analyze_horse_class_history(horse)

        if class_history['runs_analyzed'] == 0:
            return 50  # Neutral score for no history

        # Calculate suitability score (0-100)
        avg_historical = class_history['average_class_weight']

        # Horse is suited if current class is similar to historical average
        class_difference = abs(current_class_weight - avg_historical)

        # Score based on difference (lower difference = higher score)
        suitability = max(0, 100 - (class_difference * 2))

        # Adjust based on consistency
        consistency_factor = class_history['class_consistency'] / 100
        suitability *= consistency_factor

        # Bonus if horse has proven ability at this level or higher
        if class_history['highest_class'] >= current_class_weight:
            suitability = min(100, suitability * 1.2)

        return suitability

    def get_class_trend(self, horse):
        """Analyze if horse is moving up or down in class"""
        # Use absolute import instead of relative import
        from racecard.models import Run

        runs = Run.objects.filter(horse=horse).order_by('-run_date')[:4]
        class_weights = []

        for run in runs:
            if run.race_class:
                weight = self.get_class_weight(run.race_class)
                class_weights.append(weight)

        if len(class_weights) < 2:
            return "stable"  # Not enough data

        # Calculate trend (recent first)
```

```
recent = class_weights[0]
previous = sum(class_weights[1:]) / len(class_weights[1:])

if recent > previous + 5:
    return "moving_up"
elif recent < previous - 5:
    return "moving_down"
else:
    return "stable"
```