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

class ClassAnalysisService:

    def __init__(self, debug_callback=None):
        # Initialize debug callback first
        self.debug = debug_callback or (lambda x: None)  # Default to no-op
function
        # Then load class groups
        self.class_groups = self._load_class_groups()

    def _load_class_groups(self):
        """Load class groups from JSON file with debug output"""
        groups_path = os.path.join(settings.BASE_DIR, 'racecard', 'data',
'class_weights.json')
        try:
            with open(groups_path, 'r') as f:
                data = json.load(f)
                self.debug(f"✅ Loaded class groups from JSON:
{list(data['class_groups'].keys())}")
                return data['class_groups']
        except (FileNotFoundError, json.JSONDecodeError, KeyError) as e:
            self.debug(f"⚠️ Could not load class groups: {e}. Using default
groups.")
            return self._get_default_groups()

    def _get_default_groups(self):
        """Default class groups if JSON file not found"""
        return {
            "Group 1": {"min_merit": 100, "max_merit": 120, "weight": 20,
"equivalent_names": ["Group 1", "G1", "Classic", "Grade 1"]},
            "Group 2": {"min_merit": 90, "max_merit": 99, "weight": 18,
"equivalent_names": ["Group 2", "G2", "Stakes", "Grade 2"]},
            "Group 3": {"min_merit": 80, "max_merit": 89, "weight": 16,
"equivalent_names": ["Group 3", "G3", "Listed", "Grade 3"]},
            "Premier": {"min_merit": 70, "max_merit": 79, "weight": 14,
"equivalent_names": ["Premier", "MR70+", "Feature", "Premier Handicap"]},
            "Middle": {"min_merit": 60, "max_merit": 69, "weight": 12,
"equivalent_names": ["Middle", "MR60+", "Mddle", "Middle Stakes", "MR64"]},
            "Moderate": {"min_merit": 50, "max_merit": 59, "weight": 10,
"equivalent_names": ["Moderate", "MR50+", "MR55", "Handicap"]},
            "Standard": {"min_merit": 40, "max_merit": 49, "weight": 8,
"equivalent_names": ["Standard", "MR40+", "MR45", "Class 4"]},
            "Basic": {"min_merit": 30, "max_merit": 39, "weight": 6,
"equivalent_names": ["Basic", "MR30+", "MR35", "Class 5"]},
            "Maiden": {"min_merit": 0, "max_merit": 29, "weight": 4,
"equivalent_names": ["Maiden", "MP", "OM", "Novice", "Class 6"]}
        }

    def find_class_group(self, race_class):
        """Find which group a race class belongs to with debug info"""
        if not race_class:
            self.debug(f"🔍 Class analysis: No race class provided")
            return None, 0
        race_class_upper = race_class.upper().strip()
        self.debug(f"🔍 Analyzing race class: '{race_class}' ->
'{race_class_upper}'")

        # First, try to extract merit rating
        merit_match = re.search(r'MR\s*(\d+)', race_class_upper)
        if merit_match:
            merit_value = int(merit_match.group(1))
            self.debug(f"📊 Found merit rating: MR{merit_value}")

            for group_name, group_data in self.class_groups.items():
                if group_data['min_merit'] <= merit_value <=
group_data['max_merit']:
                    self.debug(f"✅ Matched MR{merit_value} to group: {group_name}
(weight: {group_data['weight']})")
                    return group_name, group_data['weight']

        # Then try to match by equivalent names
        for group_name, group_data in self.class_groups.items():
            for equivalent_name in group_data['equivalent_names']:
                if equivalent_name.upper() in race_class_upper:
                    self.debug(f"✅ Matched '{equivalent_name}' to group:
{group_name} (weight: {group_data['weight']})")
                    return group_name, group_data['weight']

        # Default to Maiden if no match found
        self.debug(f"⚠️ No specific match found for '{race_class}', defaulting to
Maiden")
        return "Maiden", self.class_groups["Maiden"]["weight"]

    def calculate_run_score(self, race_class, position):
        """Calculate a score for a single run with debug info"""
        self.debug(f"🎯 Calculating run score for class: '{race_class}', position:
{position}")

        group_name, class_weight = self.find_class_group(race_class)

        # Convert position to performance score
        try:
            pos = float(position) if position and position.isdigit() else 10
            performance_score = max(0, 100 - (pos * 20))
            self.debug(f"📈 Position {pos} -> performance score:
{performance_score}")
        except:
            performance_score = 50
            self.debug(f"⚠️ Could not parse position '{position}', using default:
50")

        # Combine class weight and performance
        run_score = (class_weight * 0.6) + (performance_score * 0.4)
        self.debug(f"🧮 Final run score: ({class_weight} × 0.6) +
({performance_score} × 0.4) = {run_score:.2f}")

        return {
            'class_group': group_name,
            'class_weight': class_weight,
            'performance_score': performance_score,
            'run_score': round(run_score, 2),
            'position': position
        }

    def analyze_horse_class_history(self, horse):
        """Analyze a horse's class history with detailed debug"""
        from racecard.models import Run

        self.debug(f"\n📊 Analyzing class history for {horse.horse_name}
(#{horse.horse_no})")

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

        if not runs:
            self.debug("ℹ️ No past runs found for this horse")
            return self._get_empty_analysis()

        self.debug(f"📅 Found {len(runs)} recent runs:")

        run_analyses = []
        total_score = 0

        for i, run in enumerate(runs, 1):
            self.debug(f"\n  Run {i}: {run.run_date} - {run.race_class or 'No
class'} - Pos: {run.position}")
            analysis = self.calculate_run_score(run.race_class, run.position)
            run_analyses.append(analysis)
            total_score += analysis['run_score']
            self.debug(f"  → Score: {analysis['run_score']:.2f}")

        avg_score = total_score / len(runs) if runs else 0
        self.debug(f"\n📈 Average run score: {avg_score:.2f}")

        # Find best performance
        best_performance = None
        for analysis in run_analyses:
            if analysis['performance_score'] >= 70:  # Good performance (top 3)
                if not best_performance or analysis['class_weight'] >
best_performance['class_weight']:
                    best_performance = analysis

        if best_performance:
            self.debug(f"⭐ Best performance: {best_performance['class_group']}
(weight: {best_performance['class_weight']}), Score:
{best_performance['run_score']:.2f}")

        return {
            'run_analyses': run_analyses,
            'average_score': round(avg_score, 2),
            'best_performance': best_performance,
            'runs_analyzed': len(runs),
            'recent_class': run_analyses[0]['class_group'] if run_analyses else
None,
            'recent_performance': run_analyses[0]['performance_score'] if
run_analyses else 0
        }

    def calculate_class_suitability(self, horse, current_race):
        """Calculate how suitable the horse is for the current race class"""
        current_group, current_weight =
self.find_class_group(current_race.race_class)
```

```
        class_history = self.analyze_horse_class_history(horse)

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

        # Base suitability based on average performance
        suitability = class_history['average_score']

        # Bonus if horse has proven ability at this level or higher
        if class_history['best_performance']:
            if class_history['best_performance']['class_weight'] >= current_weight:
                suitability = min(100, suitability * 1.2)  # 20% bonus
                self.debug(f"🎯 Bonus: Proven ability at this level or higher
(+20%)")

        # Penalty if moving up significantly
        avg_class_weight = sum(analysis['class_weight'] for analysis in
class_history['run_analyses']) / len(class_history['run_analyses'])
        if current_weight > avg_class_weight + 2:
            suitability *= 0.8  # 20% penalty
            self.debug(f"⚠️ Penalty: Moving up significantly in class (-20%)")

        final_score = min(100, max(0, suitability))
        self.debug(f"🏁 Final class suitability score: {final_score:.2f}")

        return final_score

    def get_class_trend(self, horse):
        """Analyze if horse is moving up or down in class"""
        class_history = self.analyze_horse_class_history(horse)

        if class_history['runs_analyzed'] < 2:
            return "stable"

        recent = class_history['run_analyses'][0]
        previous_avg = sum(analysis['class_weight'] for analysis in
class_history['run_analyses'][1:]) / (class_history['runs_analyzed'] - 1)

        if recent['class_weight'] > previous_avg + 1:
            return "moving_up"
        elif recent['class_weight'] < previous_avg - 1:
            return "moving_down"
        else:
            return "stable"

    def _get_empty_analysis(self):
        return {
            'run_analyses': [],
            'average_score': 0,
            'best_performance': None,
            'runs_analyzed': 0,
            'recent_class': None,
            'recent_performance': 0
        }

    def get_class_weight(self, race_class):
        """Get the weight for a given race class (backward compatibility)"""
        _, weight = self.find_class_group(race_class)
        return weight
```