```
#!/usr/bin/env python3
"""
CV FMV Calculator - Production Level Script (Fully Corrected)
Calculates Fair Market Value (FMV) for all doctor entries from CVdump.csv
Based on scoring_criteria.xlsx and OUS FMV Rates
Author: AI Assistant
Version: 2.0 - Fully Corrected with 100% Reliability Enhancements
"""

import pandas as pd
import os
import sys
import logging
import unicodedata
from datetime import datetime
from typing import Dict, List, Optional, Tuple
from difflib import get_close_matches
import traceback

# ================================================================================
# CONFIGURATION & LOGGING SETUP
# ================================================================================

# Setup logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler('cv_fmv_calculator.log', encoding='utf-8'),
        logging.StreamHandler(sys.stdout)
    ]
)
logger = logging.getLogger(__name__)

# File paths
CVDUMP_FILE = "CVdump.csv"
SCORING_CRITERIA_FILE = "scoring_criteria.xlsx"
OUTPUT_FILE = f"CV_FMV_Results_{datetime.now().strftime('%Y%m%d_%H%M%S')}.xlsx"
QUALITY_REPORT_FILE = f"Quality_Report_{datetime.now().strftime('%Y%m%d_%H%M%S')}.txt"

# ================================================================================
# UTILITY FUNCTIONS
# ================================================================================

def normalize_text(text):
    """Normalize text by removing special characters and extra spaces"""
    if pd.isna(text) or text == '' or text == 'nan':
        return ''
    # Convert to string and normalize unicode
    text = str(text)
    text = unicodedata.normalize('NFKD', text)

    # Replace various line breaks with standard space
    text = text.replace('\r\n', ' ').replace('\r', ' ').replace('\n', ' ')

    # Replace non-breaking spaces with regular spaces
    text = text.replace('\xa0', ' ')

    # Remove extra whitespace
    text = ' '.join(text.split())

    return text.strip()

def fuzzy_match_score(response, scoring_dict, threshold=0.85):
    """Get score with fuzzy matching to handle text variations"""
    if not response or response == '':
        logger.warning("Empty response provided for scoring")
        return 0, False

    response_normalized = normalize_text(response)

    # Try exact match first (normalized)
    for key, value in scoring_dict.items():
        if normalize_text(key) == response_normalized:
            return value, True

    # Try fuzzy match
    normalized_keys = {normalize_text(k): k for k in scoring_dict.keys()}
    matches = get_close_matches(response_normalized, normalized_keys.keys(), n=1,
cutoff=threshold)

    if matches:
        matched_normalized = matches[0]
        original_key = normalized_keys[matched_normalized]
        score = scoring_dict[original_key]
        logger.info(f"Fuzzy matched: '{response}' → '{original_key}' (score: {score})")
        return score, True

    # No match found
    logger.warning(f"No match found for response: '{response}'")
    return 0, False

# ============================================================================
# SCORING CRITERIA LOOKUP DICTIONARIES
# ============================================================================

def create_scoring_lookup():
    """Create comprehensive scoring lookup dictionaries"""

    # Years of Experience scoring
    years_experience_scores = {
        "1-2 years of experience": 0,
        "3-7 years of experience": 2,
        "8-14 years of experience": 4,
        "15+ years of experience": 6
    }

    # Clinical Experience scoring
    clinical_experience_scores = {
        "Minimal patient interactions and predominantly administrative/academic work": 0,
        "Less than half the time spent with patients in clinical setting and higher focus on
academic/administrative work": 2,
        "Equal amount of time spent with patients in clinical setting and equal amount of time
spent in academic/administrative work": 4,
        "Significant time spent with patients in clinical setting and minimal time spent in
academic/administrative work": 6
    }

    # Leadership position scoring
    leadership_scores = {
        "Not applicable, as not a part of any society or leadership roles in hospital": 0,
        "1-2 years in a leadership position(s) eg. HOD of a particular speciality in Hospital or
other Patient Care Setting and/or serving as a President, Vice president, Secretary,Treasurer,
Board member in a Professional or Scientific Society.": 2,
        "3-7 years in a leadership position(s) eg HOD of a particular speciality in Hospital or
other Patient Care Setting and/or serving as a national/regional leader in a Professional or
Scientific Society.": 4,
        "8 or more years in a leadership position(s) eg HOD for a specialty in Hospital or other
Patient Care Setting and/or serving as an international leader in a Professional or Scientific
Society.": 6
    }

    # Geographical Reach scoring
    geographical_reach_scores = {
        "Local Influence": 0,
        "National Influence": 2,
        "Multi-Country Influence": 4,
        "Global/Worldwide Influence": 6
    }

    # Highest Academic Position scoring
    academic_position_scores = {
        "None or N/A": 0,
        "Professor (including Associate / Assistant Professor)": 2,
        "Professor or Adjunct/Additional/Emeritus Professor": 4,
        "Department Chair/ HOD (or similar position)": 6
    }
    # Additional Educational Level scoring
    educational_level_scores = {
        "None or N/A": 0,
        "1 Additional degree, fellowship, or advanced training certification.": 2,
        "2 Additional degrees, fellowship, or advanced training certification.": 4,
        "3 or More Additional degrees, fellowship, or advanced training certification.": 6
    }

    # Research Experience scoring
    research_experience_scores = {
        "None or N/A": 0,
        "Participation as an Investigator or Sub-Investigator in 1 to 4 clinical trials or
research studies.": 2,
        "Participation as an Investigator or Sub-Investigator in 5 to 9 clinical trials or
research studies.": 4,
        "Participation as an Investigator of Sub-Investigator in 10 or more clinical trials or
research studies or Principal Investigator for two or more clinical trials or research studies
or serving as the Principal Investigator for a clinical trial or research study that led to
important medical innovations or significant medical technology breakthroughs.": 6
    }

    # Publication Experience scoring
    publication_experience_scores = {
        "None or N/A": 0,
        "Co-authorship or participation as contributing author on 1 to 4 publications.": 2,
        "First authorship (if known) on 1 to 5 publications and/or co-authorship or
participation as contributing author on 6 to 10 publications": 4,
        "First authorship (if known) on 6 or more publications and/or co-authorship or
participation as contributing author on 11 or more publications": 6
    }

    # Speaking Experience scoring
    speaking_experience_scores = {
        "Local speaking engagements and the scientific work done for the specialty is near to
the practice location": 0,
        "Most of the speaking engagements are directed nationally for the conferences, symposia
or national webinars in the designated specialty and the scientific work done is not restricted
for the local audience": 2,
        "The speaking experiences are not restricted nationally but to a group of specified
countries and the scientific work is directed to the same group of countries": 4,
        "The speaking engagements and the scinetific work carried out is across the globe": 6
    }

    return {
        "years_experience": years_experience_scores,
        "clinical_experience": clinical_experience_scores,
        "leadership": leadership_scores,
        "geographical_reach": geographical_reach_scores,
        "academic_position": academic_position_scores,
        "educational_level": educational_level_scores,
        "research_experience": research_experience_scores,
        "publication_experience": publication_experience_scores,
        "speaking_experience": speaking_experience_scores
    }

# ===============================================================================
# FMV RATES LOADING
# ===============================================================================

def load_fmv_rates():
    """Load FMV rates from OUS FMV Rates sheet"""
    try:
        rates_df = pd.read_excel(SCORING_CRITERIA_FILE, sheet_name="OUS FMV Rates", header=1)

        # Filter for India rates
        india_rates = rates_df[rates_df['Country'] == 'India'].copy()

        # Create specialty to rates mapping
        specialty_rates = {}
        for _, row in india_rates.iterrows():
            specialty = str(row['HCP Specialty']).strip()
            if pd.notna(specialty) and specialty != '' and specialty != 'nan':
                specialty_rates[specialty] = {
                    'Tier 1': int(row['Tier 1']) if pd.notna(row['Tier 1']) else 0,
                    'Tier 2': int(row['Tier 2']) if pd.notna(row['Tier 2']) else 0,
                    'Tier 3': int(row['Tier 3']) if pd.notna(row['Tier 3']) else 0,
                    'Tier 4': int(row['Tier 4']) if pd.notna(row['Tier 4']) else 0
                }

        logger.info(f"Loaded FMV rates for {len(specialty_rates)} specialties")

        # Log available specialties for debugging
        logger.info(f"Available specialties: {list(specialty_rates.keys())}")

        return specialty_rates
    except Exception as e:
        logger.error(f"Error loading FMV rates: {str(e)}")
        logger.error(traceback.format_exc())
        raise

# ===============================================================================
# DATA VALIDATION FUNCTIONS
# ===============================================================================

def validate_doctor_data(row):
    """Validate that doctor has minimum required data"""
    required_fields = {
        "HCP Name": row.get("HCP Name", ""),
        "HCP Email": row.get("HCP Email", ""),
        "Specialty / Super Specialty": row.get("Specialty / Super Specialty", "")
    }

    missing_fields = []
    for field_name, value in required_fields.items():
        normalized_value = normalize_text(str(value))
        if not normalized_value or normalized_value == 'nan':
            missing_fields.append(field_name)

    if missing_fields:
        logger.warning(f"Doctor {row.get('HCP Name', 'Unknown')} missing: {missing_fields}")
        return False, missing_fields

    return True, []

# ==============================================================================
# SCORING FUNCTIONS
# ==============================================================================

def calculate_individual_scores(row, scoring_lookup):
    """Calculate individual scores for each criterion with enhanced error tracking"""
    scores = {}
    unmatched_fields = []

    # Define field mappings
    field_mappings = [
        ("score_1", "years_experience", "Years of experience in the Specialty / Super Specialty?
\n"),
        ("score_2", "clinical_experience", "Clinical Experience: i.e. Time Spent with
Patients?"),
        ("score_3", "leadership", "Leadership position(s) in a Professional or Scientific
Society and/or leadership position(s) in Hospital or other Patient Care Settings (e.g.
Department Head or Chief, Medical Director, Lab Direct..."),
        ("score_4", "geographical_reach", "Geographic influence as a Key Opinion Leader."),
        ("score_5", "academic_position", "Highest Academic Position Held in past 10 years"),
        ("score_6", "educational_level", "Additional Educational Level "),
        ("score_7", "research_experience", "Research Experience (e.g., industry-sponsored
research, investigator-initiated research, other research) in past 10 years"),
        ("score_8", "publication_experience", "Publication experience in the past 10 years"),
        ("score_9", "speaking_experience", "Speaking experience (professional, academic,
scientific, or media experience) in the past 10 years.")
    ]

    for score_key, lookup_key, field_name in field_mappings:
        response = str(row.get(field_name, ""))
        score, matched = fuzzy_match_score(response, scoring_lookup[lookup_key])

        if not matched and response and response != 'nan':
            unmatched_fields.append({
                'field': field_name,
                'response': response,
                'score_assigned': score
            })

        scores[score_key] = score

    # Calculate total score
    scores["total_score"] = sum([scores[f"score_{i}"] for i in range(1, 10)])

    return scores, unmatched_fields

def determine_tier(total_score):
    """Determine tier based on total score"""
    if total_score <= 13:
        return "Tier 1"
    elif total_score <= 26:
        return "Tier 2"
    elif total_score <= 40:
        return "Tier 3"
    else:
        return "Tier 4"

def calculate_fmv_amount(specialty, tier, fmv_rates):
    """Calculate FMV amount based on specialty and tier with fuzzy matching"""
    specialty_clean = normalize_text(specialty)

    # Try exact match first (normalized)
    for rate_specialty, rates in fmv_rates.items():
        if normalize_text(rate_specialty) == specialty_clean:
            amount = rates.get(tier, 0)
            logger.info(f"FMV matched: '{specialty}' → '{rate_specialty}' → {tier}: ₹{amount}")
            return amount

    # Try fuzzy match on specialty
    normalized_specialties = {normalize_text(k): k for k in fmv_rates.keys()}
    matches = get_close_matches(specialty_clean, normalized_specialties.keys(), n=1,
cutoff=0.80)

    if matches:
        matched_normalized = matches[0]
        original_specialty = normalized_specialties[matched_normalized]
        amount = fmv_rates[original_specialty].get(tier, 0)
        logger.info(f"FMV fuzzy matched: '{specialty}' → '{original_specialty}' → {tier}: ₹
{amount}")
        return amount

    # Use conservative default rates if no match
    default_rates = {
        "Tier 1": 5000,
        "Tier 2": 7000,
        "Tier 3": 9000,
        "Tier 4": 12000
    }

    amount = default_rates.get(tier, 5000)
    logger.warning(f"No FMV rate found for specialty '{specialty}', using default: {tier} = ₹
{amount}")

    return amount

# ============================================================================
# DATA PROCESSING FUNCTIONS
# ============================================================================

def load_cvdump_data():
    """Load and clean CVdump.csv data"""
    try:
        logger.info("Loading CVdump.csv data...")

        # Try different encodings
        encodings = ['utf-8', 'latin-1', 'cp1252', 'iso-8859-1']
        df = None

        for encoding in encodings:
            try:
                df = pd.read_csv(CVDUMP_FILE, encoding=encoding)
                logger.info(f"Successfully loaded CVdump.csv with {encoding} encoding")
                break
            except UnicodeDecodeError:
                continue
            except Exception as e:
                logger.error(f"Error with {encoding}: {str(e)}")
                continue

        if df is None:
            raise Exception("Could not load CVdump.csv with any supported encoding")

        logger.info(f"Initial records loaded: {len(df)}")

        # Clean email addresses
        df["HCP Email"] = df["HCP Email"].astype(str).str.strip().str.lower()

        # Remove rows with invalid emails
        initial_count = len(df)
        df = df[df["HCP Email"] != "nan"]
        df = df[df["HCP Email"] != ""]
        df = df[df["HCP Email"].str.contains('@', na=False)]

        removed = initial_count - len(df)
        if removed > 0:
            logger.warning(f"Removed {removed} rows with invalid emails")

        logger.info(f"Valid records after email cleaning: {len(df)}")

        return df
    except Exception as e:
        logger.error(f"Error loading CVdump data: {str(e)}")
        logger.error(traceback.format_exc())
        raise

def deduplicate_cvdump(df):
    """Remove duplicate entries, keeping the most recent if timestamp available"""
    try:
        initial_count = len(df)

        if "Start time" in df.columns:
            logger.info("Deduplicating based on Start time...")

            # Parse datetime with multiple format attempts
            df["Start time_parsed"] = pd.to_datetime(df["Start time"], errors='coerce')

            # Sort by email and time
            df = df.sort_values(["HCP Email", "Start time_parsed"], na_position='last')

            # Keep latest entry per email
            df = df.drop_duplicates("HCP Email", keep="last")

            # Drop the parsed column
            df = df.drop("Start time_parsed", axis=1)
        else:
            logger.info("No Start time column found, deduplicating by keeping first
occurrence...")
            df = df.drop_duplicates("HCP Email", keep="first")

        removed = initial_count - len(df)
        logger.info(f"After deduplication: {len(df)} unique doctors (removed {removed}
duplicates)")

        return df
    except Exception as e:
        logger.error(f"Error in deduplication: {str(e)}")
        return df

def process_doctor_data(df, scoring_lookup, fmv_rates):
    """Process each doctor's data and calculate FMV"""
    results = []
    quality_issues = {
        'validation_failures': [],
        'unmatched_responses': [],
        'missing_fmv': []
    }

    logger.info(f"Processing {len(df)} doctors...")

    for index, row in df.iterrows():
        try:
            # Validate doctor data
            is_valid, missing_fields = validate_doctor_data(row)
            if not is_valid:
                quality_issues['validation_failures'].append({
                    'doctor': row.get('HCP Name', 'Unknown'),
                    'email': row.get('HCP Email', 'Unknown'),
                    'missing_fields': missing_fields
                })
                logger.warning(f"Skipping doctor due to validation failure: {row.get('HCP Name', 
'Unknown')}")
                continue

            # Calculate individual scores
            scores, unmatched_fields = calculate_individual_scores(row, scoring_lookup)
            total_score = scores["total_score"]

            # Track unmatched responses
            if unmatched_fields:
                quality_issues['unmatched_responses'].append({
                    'doctor': row.get('HCP Name', ''),
                    'email': row.get('HCP Email', ''),
                    'unmatched': unmatched_fields
                })

            # Determine tier
            tier = determine_tier(total_score)

            # Get specialty
            specialty = normalize_text(str(row.get("Specialty / Super Specialty", "")))

            # Calculate FMV amount
            fmv_amount = calculate_fmv_amount(specialty, tier, fmv_rates)

            # Track missing FMV rates
            if fmv_amount <= 5000 and tier != "Tier 1":  # Default rate used
                quality_issues['missing_fmv'].append({
                    'doctor': row.get('HCP Name', ''),
                    'specialty': specialty,
                    'tier': tier
                })

            # Create result record matching FMV_Calculator_Updated.xlsx structure
            result = {
                "i": index + 1,  # Sequential number
                "HCP Name": row.get("HCP Name", ""),
                "Years of experience in the Specialty / Super Specialty?_x000D_\n":
row.get("Years of experience in the Specialty / Super Specialty?\n", ""),
                "Clinical Experience: i.e. Time Spent with Patients?": row.get("Clinical
Experience: i.e. Time Spent with Patients?", ""),
                "Leadership position(s) in a Professional or Scientific Society and/or
leadership position(s) in Hospital or other Patient Care Settings (e.g. Department Head or
Chief, Medical Director, Lab Direct...": row.get("Leadership position(s) in a Professional or
Scientific Society and/or leadership position(s) in Hospital or other Patient Care Settings
(e.g. Department Head or Chief, Medical Director, Lab Direct...", ""),
                "Geographic influence as a Key Opinion Leader.": row.get("Geographic influence
as a Key Opinion Leader.", ""),
                "Highest Academic Position Held in past 10 years": row.get("Highest Academic
Position Held in past 10 years", ""),
                "Additional Educational Level": row.get("Additional Educational Level ", ""),
                "Research Experience (e.g., industry-sponsored research, investigator-initiated
research, other research) in past 10 years": row.get("Research Experience (e.g., industry-
sponsored research, investigator-initiated research, other research) in past 10 years", ""),
                "Publication experience in the past 10 years": row.get("Publication experience
in the past 10 years", ""),
                "Speaking experience (professional, academic, scientific, or media experience)
in the past 10 years.": row.get("Speaking experience (professional, academic, scientific, or
media experience) in the past 10 years.", ""),
                "Score based on selection mentioned criteria": total_score,
                "Score 1": scores["score_1"],
                "Score 2": scores["score_2"],
                "Score 3": scores["score_3"],
                "Score 4": scores["score_4"],
                "Score 5": scores["score_5"],
                "Score 6": scores["score_6"],
                "Score 7": scores["score_7"],
                "Score 8": scores["score_8"],
                "Score 9": scores["score_9"],
                "Range": f"{total_score}-{total_score}",
                "Tier": tier,
                "Rate of Honorarium": fmv_amount,
                "Specialty / Super Specialty": specialty,
                "HCP Email": row.get("HCP Email", ""),
                "Educational Qualification": row.get("Educational Qualification", "")
            }

            results.append(result)

            # Log progress every 50 doctors
            if (index + 1) % 50 == 0:
                logger.info(f"Processed {index + 1} doctors...")

        except Exception as e:
            logger.error(f"Error processing doctor {row.get('HCP Name', 'Unknown')} (row
{index}): {str(e)}")
```

```
            logger.error(traceback.format_exc())
            continue

    logger.info(f"Successfully processed {len(results)} doctors")

    return results, quality_issues

# ================================================================================
# QUALITY REPORT GENERATION
# ================================================================================

def generate_quality_report(results, quality_issues):
    """Generate comprehensive data quality report"""
    report_lines = []
    report_lines.append("=" * 80)
    report_lines.append("DATA QUALITY REPORT")
    report_lines.append("=" * 80)
    report_lines.append(f"Generated: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
    report_lines.append("")

    # Overall statistics
    report_lines.append("OVERALL STATISTICS")
    report_lines.append("-" * 80)
    report_lines.append(f"Total Doctors Processed: {len(results)}")
    report_lines.append(f"Validation Failures: {len(quality_issues['validation_failures'])}")
    report_lines.append(f"Doctors with Unmatched Responses:
{len(quality_issues['unmatched_responses'])}")
    report_lines.append(f"Doctors with Default FMV Rates: {len(quality_issues['missing_fmv'])}")
    report_lines.append("")

    # Score distribution
    if results:
        report_lines.append("SCORE DISTRIBUTION")
        report_lines.append("-" * 80)
        scores = [r['Score based on selection mentioned criteria'] for r in results]
        report_lines.append(f"Minimum Score: {min(scores)}")
        report_lines.append(f"Maximum Score: {max(scores)}")
        report_lines.append(f"Average Score: {sum(scores)/len(scores):.2f}")
        report_lines.append(f"Median Score: {sorted(scores)[len(scores)//2]}")

        # Zero scores
        zero_scores = [r for r in results if r['Score based on selection mentioned criteria'] ==
0]
        report_lines.append(f"Doctors with Zero Score: {len(zero_scores)}")
        report_lines.append("")

        # Tier distribution
        report_lines.append("TIER DISTRIBUTION")
        report_lines.append("-" * 80)
        tier_counts = {}
        for r in results:
            tier = r['Tier']
            tier_counts[tier] = tier_counts.get(tier, 0) + 1

        for tier in ['Tier 1', 'Tier 2', 'Tier 3', 'Tier 4']:
            count = tier_counts.get(tier, 0)
            percentage = (count / len(results) * 100) if results else 0
            report_lines.append(f"{tier}: {count} doctors ({percentage:.1f}%)")
        report_lines.append("")

        # FMV statistics
        report_lines.append("FMV STATISTICS")
        report_lines.append("-" * 80)
        fmv_amounts = [r['Rate of Honorarium'] for r in results]
        total_fmv = sum(fmv_amounts)
        report_lines.append(f"Total FMV Amount: ₹{total_fmv:,}")
        report_lines.append(f"Average FMV per Doctor: ₹{total_fmv/len(results):,.2f}")
        report_lines.append(f"Minimum FMV: ₹{min(fmv_amounts):,}")
        report_lines.append(f"Maximum FMV: ₹{max(fmv_amounts):,}")
        report_lines.append("")

    # Validation failures detail
    if quality_issues['validation_failures']:
        report_lines.append("VALIDATION FAILURES (Sample - First 10)")
        report_lines.append("-" * 80)
        for i, failure in enumerate(quality_issues['validation_failures'][:10]):
            report_lines.append(f"{i+1}. {failure['doctor']} ({failure['email']})")
            report_lines.append(f"   Missing: {', '.join(failure['missing_fields'])}")

        if len(quality_issues['validation_failures']) > 10:
            report_lines.append(f"   ... and {len(quality_issues['validation_failures']) - 10}
more")
        report_lines.append("")

    # Unmatched responses summary
    if quality_issues['unmatched_responses']:
        report_lines.append("UNMATCHED RESPONSES SUMMARY")
        report_lines.append("-" * 80)
        report_lines.append(f"Total doctors with unmatched responses:
{len(quality_issues['unmatched_responses'])}")

        # Count unmatched by field
        field_counts = {}
        for issue in quality_issues['unmatched_responses']:
            for unmatched in issue['unmatched']:
                field = unmatched['field']
                field_counts[field] = field_counts.get(field, 0) + 1

        report_lines.append("\nMost problematic fields:")
        for field, count in sorted(field_counts.items(), key=lambda x: x[1], reverse=True)[:5]:
            report_lines.append(f"  - {field}: {count} unmatched responses")
        report_lines.append("")

    # Missing FMV rates
    if quality_issues['missing_fmv']:
        report_lines.append("MISSING FMV RATES (Using Defaults)")
        report_lines.append("-" * 80)
        specialty_counts = {}
        for issue in quality_issues['missing_fmv']:
            specialty = issue['specialty']
            specialty_counts[specialty] = specialty_counts.get(specialty, 0) + 1

        report_lines.append("Specialties using default rates:")
        for specialty, count in sorted(specialty_counts.items(), key=lambda x: x[1],
reverse=True):
            report_lines.append(f"  - {specialty}: {count} doctors")
        report_lines.append("")

    report_lines.append("=" * 80)
    report_lines.append("END OF REPORT")
    report_lines.append("=" * 80)

    report_text = "\n".join(report_lines)

    # Log to console
    for line in report_lines:
        logger.info(line)

    # Save to file
    try:
        with open(QUALITY_REPORT_FILE, 'w', encoding='utf-8') as f:
            f.write(report_text)
        logger.info(f"Quality report saved to: {QUALITY_REPORT_FILE}")
    except Exception as e:
        logger.error(f"Error saving quality report: {str(e)}")

    return report_text

# ============================================================================
# FILE SAVING WITH TRANSACTION-LIKE BEHAVIOR
# ============================================================================

def save_results(results):
    """Save results to Excel file with rollback capability"""
    try:
        if not results:
            logger.error("No results to save!")
            return None

        results_df = pd.DataFrame(results)
        # Create temporary file first
        temp_file = OUTPUT_FILE + ".tmp"

        # Save to temporary file
        logger.info(f"Saving to temporary file: {temp_file}")
        results_df.to_excel(temp_file, sheet_name='Sheet1', index=False, engine='openpyxl')

        # Verify temp file was created
        if not os.path.exists(temp_file):
            raise Exception("Failed to create temporary file")

        # Verify file size is reasonable
        file_size = os.path.getsize(temp_file)
        if file_size < 1000:  # Less than 1KB is suspicious
            raise Exception(f"Output file suspiciously small: {file_size} bytes")

        # Move temp file to final location
        if os.path.exists(OUTPUT_FILE):
            os.remove(OUTPUT_FILE)
        os.rename(temp_file, OUTPUT_FILE)

        logger.info(f"Results successfully saved to: {OUTPUT_FILE}")
        logger.info(f"File size: {os.path.getsize(OUTPUT_FILE):,} bytes")

        return OUTPUT_FILE

    except Exception as e:
        logger.error(f"Error saving results: {str(e)}")
        logger.error(traceback.format_exc())

        # Clean up temp file if it exists
        temp_file = OUTPUT_FILE + ".tmp"
        if os.path.exists(temp_file):
            try:
                os.remove(temp_file)
                logger.info("Cleaned up temporary file")
            except:
                pass

        raise

# ================================================================================
# SUMMARY DISPLAY
# ================================================================================

def display_summary(results, quality_issues):
    """Display user-friendly summary"""
    print("\n" + "=" * 80)
    print("CV FMV CALCULATOR - EXECUTION SUMMARY")
    print("=" * 80)

    if results:
        # Basic stats
        total_fmv = sum(r['Rate of Honorarium'] for r in results)
        avg_score = sum(r['Score based on selection mentioned criteria'] for r in results) / 
len(results)

        print(f"\n✅ Successfully processed {len(results)} doctors")
        print(f"\n📊 STATISTICS:")
        print(f"   Average Score: {avg_score:.2f}")
        print(f"   Total FMV Amount: ₹{total_fmv:,}")
        print(f"   Average FMV per Doctor: ₹{total_fmv/len(results):,.2f}")

        # Tier distribution
        tier_counts = {}
        for r in results:
            tier = r['Tier']
            tier_counts[tier] = tier_counts.get(tier, 0) + 1

        print(f"\n🎯 TIER DISTRIBUTION:")
        for tier in ['Tier 1', 'Tier 2', 'Tier 3', 'Tier 4']:
            count = tier_counts.get(tier, 0)
            percentage = (count / len(results) * 100) if results else 0
            print(f"   {tier}: {count} doctors ({percentage:.1f}%)")

        # Quality issues
        print(f"\n⚠️  QUALITY ISSUES:")
        print(f"   Validation Failures: {len(quality_issues['validation_failures'])}")
        print(f"   Unmatched Responses: {len(quality_issues['unmatched_responses'])}")
        print(f"   Default FMV Rates Used: {len(quality_issues['missing_fmv'])}")

        # Top specialties
        specialty_counts = {}
        for r in results:
            spec = r['Specialty / Super Specialty']
            specialty_counts[spec] = specialty_counts.get(spec, 0) + 1

        if specialty_counts:
            print(f"\n🏥 TOP 5 SPECIALTIES:")
            top_specialties = sorted(specialty_counts.items(), key=lambda x: x[1], reverse=True)
[:5]
            for i, (spec, count) in enumerate(top_specialties, 1):
                print(f"   {i}. {spec}: {count} doctors")

        # Files generated
        print(f"\n📁 FILES GENERATED:")
        print(f"   Results: {OUTPUT_FILE}")
        print(f"   Quality Report: {QUALITY_REPORT_FILE}")
        print(f"   Log File: cv_fmv_calculator.log")
    else:
        print("\n❌ No results generated - check logs for errors")

    print("\n" + "=" * 80)

# ==============================================================================
# MAIN EXECUTION
# ==============================================================================

def main():
    """Main execution function"""
    start_time = datetime.now()

    try:
        logger.info("=" * 80)
        logger.info("STARTING CV FMV CALCULATOR - Version 2.0 (Fully Corrected)")
        logger.info("=" * 80)
        logger.info(f"Start time: {start_time.strftime('%Y-%m-%d %H:%M:%S')}")
        logger.info("")

        # Verify files exist
        logger.info("Verifying input files...")
        if not os.path.exists(CVDUMP_FILE):
            raise FileNotFoundError(f"CVdump file not found: {CVDUMP_FILE}")
        if not os.path.exists(SCORING_CRITERIA_FILE):
            raise FileNotFoundError(f"Scoring criteria file not found: {SCORING_CRITERIA_FILE}")
        logger.info("✓ All input files found")
        logger.info("")

        # Load scoring criteria
        logger.info("Loading scoring criteria...")
        scoring_lookup = create_scoring_lookup()
        logger.info("✓ Scoring criteria loaded")
        logger.info("")

        # Load FMV rates
        logger.info("Loading FMV rates...")
        fmv_rates = load_fmv_rates()
        logger.info("✓ FMV rates loaded")
        logger.info("")

        # Load CVdump data
        logger.info("Loading CVdump data...")
        cvdump_df = load_cvdump_data()
        logger.info("✓ CVdump data loaded")
        logger.info("")

        # Deduplicate data
        logger.info("Deduplicating data...")
        cvdump_df = deduplicate_cvdump(cvdump_df)
        logger.info("✓ Deduplication complete")
        logger.info("")

        # Process doctor data
        logger.info("Processing doctor data and calculating FMV...")
        results, quality_issues = process_doctor_data(cvdump_df, scoring_lookup, fmv_rates)
        logger.info("✓ Processing complete")
        logger.info("")

        # Generate quality report
        logger.info("Generating quality report...")
        generate_quality_report(results, quality_issues)
        logger.info("✓ Quality report generated")
        logger.info("")

        # Save results
        logger.info("Saving results...")
        output_file = save_results(results)
        logger.info("✓ Results saved")
        logger.info("")

        # Calculate execution time
        end_time = datetime.now()
        execution_time = (end_time - start_time).total_seconds()

        logger.info("=" * 80)
        logger.info("CV FMV CALCULATOR COMPLETED SUCCESSFULLY")
        logger.info("=" * 80)
        logger.info(f"Execution time: {execution_time:.2f} seconds")
        logger.info("")

        # Display summary
        display_summary(results, quality_issues)

        return True

    except Exception as e:
        logger.error("=" * 80)
        logger.error("CRITICAL ERROR - PROCESS FAILED")
        logger.error("=" * 80)
        logger.error(f"Error: {str(e)}")
        logger.error(f"Traceback:\n{traceback.format_exc()}")

        # Calculate execution time even on failure
        end_time = datetime.now()
        execution_time = (end_time - start_time).total_seconds()
        logger.error(f"Failed after {execution_time:.2f} seconds")

        print("\n❌ CV FMV Calculator failed - check cv_fmv_calculator.log for details")
        return False

# ==============================================================================
# ENTRY POINT - MUST BE AT THE END
# ==============================================================================

if __name__ == "__main__":
    success = main()
    sys.exit(0 if success else 1)
```