```python
# -*- coding: utf-8 -*-

import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from datetime import datetime
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.ensemble import RandomForestRegressor


import os
from django.conf import settings


#
----------------------------------------------------------------------
----------------------

def load_datasets(spring_path, fall_path):
    try:
        # Load the datasets
        spring_df = pd.read_csv(spring_path)
        fall_df = pd.read_csv(fall_path)
    except FileNotFoundError as e:
        print(f"Error loading data: {e}")
        return None, None

    # Add term columns
    spring_df['Term'] = 'Spring 2023'
    fall_df['Term'] = 'Fall 2023'

    return spring_df, fall_df

#
----------------------------------------------------------------------
----------------------

def merge_datasets(spring_df, fall_df):
    if spring_df is None or fall_df is None:
        return None
```

```
    # Merge the datasets
    combined_df = pd.concat([spring_df, fall_df],
ignore_index=True)

    # Clean and process data
    combined_df = clean_data(combined_df)

    file_path = "ml_api/classify-datasets/combined_dataset.csv"
    save_combined_data(combined_df, path=file_path)

    return combined_df

#
------------------------------------------------------------------
---------------------

def clean_data(df):
    # Standardize time format
    df['begin'] = df['begin'].apply(military_to_standard)
    df['end'] = df['end'].apply(military_to_standard)

    # Convert 'day' and other categorical data to appropriate
types
    df['day'] = df['day'].astype('category')

    # Handle missing values
    df.fillna({'prereq1': 'None', 'prereq2': 'None'},
inplace=True)

    # Ensure numeric columns are of type int where applicable
    numeric_cols = ['num', 'section', 'class_num', 'units']
    df[numeric_cols] = df[numeric_cols].apply(pd.to_numeric,
errors='coerce').astype('Int64')

    return df

#
------------------------------------------------------------------
---------------------

#
------------------------------------------------------------------
---------------------

def military_to_standard(time_int):
    # Ensure the string is zero-padded to 4 characters
    time_str = str(time_int).zfill(4)
```

```
    # Insert a colon between hours and minutes
    if ':' not in time_str:
       time_str = time_str[:2] + ':' + time_str[2:]
    # Convert to standard time format using datetime
    return datetime.strptime(time_str, "%H:%M").strftime("%I:%M
%p")

#
------------------------------------------------------------------
--------------------

def save_combined_data(df, path):
    if df is not None:
        df.to_csv(path, index=False)

#
------------------------------------------------------------------
--------------------

def perform_eda(data, output_dir='/Users/alicevang/Desktop/696-
proj/classify-project/front-end/src/data'):
    """Perform exploratory data analysis and return results
including paths to plots."""
    if data is None:
        return None

    # Calculate summary statistics and convert to dict
    summary_statistics = data.describe().to_dict()

    # Calculate distributions and convert to dict
    course_distribution = data['day'].value_counts().to_dict()
    start_times_distribution =
data['begin'].value_counts().to_dict()

    # Generate and save plots, then get the file paths
    classes_by_day_plot_path = plot_classes_by_day(data,
output_dir)
    class_start_times_plot_path = plot_class_start_times(data,
output_dir)

    # Compile all EDA results, including paths to plots
    results = {
        "summary_statistics": summary_statistics,
        "course_distribution": course_distribution,
        "start_times_distribution": start_times_distribution,
        "classes_by_day_plot": classes_by_day_plot_path,
        "class_start_times_plot": class_start_times_plot_path,
```

```
    }

    return results

#
------------------------------------------------------------------
--------------------

def plot_classes_by_day(data, output_dir='/Users/alicevang/
Desktop/696-proj/classify-project/front-end/src/data'):

    plt.figure(figsize=(10, 6))
    sns.countplot(x='day', data=data, order=['MW', 'TuTh', 'M',
'Tu', 'W', 'Th', 'Fr', 'Sa',])
    plt.title('Distribution of Classes Across Days of the Week')
    plt.xlabel('Day of the Week')
    plt.ylabel('Number of Classes')

    plot_path = os.path.join(output_dir, 'classes_by_day.png')
    plt.savefig(plot_path)
    plt.close()

    return plot_path

#
------------------------------------------------------------------
--------------------

def plot_class_start_times(data, output_dir='/Users/alicevang/
Desktop/696-proj/classify-project/front-end/src/data'):

    plt.figure(figsize=(14, 7))
    sns.countplot(x='begin', data=data,
order=sorted(data['begin'].unique()))
    plt.title('Distribution of Class Start Times')
    plt.xticks(rotation=90)
    plt.xlabel('Class Start Time')
    plt.ylabel('Number of Classes')

    plot_path = os.path.join(output_dir,
'classes_start_times.png')
    plt.savefig(plot_path)
    plt.close()

    return plot_path

#
```

```
# ----------------------------------------------------------------------
# ----------------------

def data_analysis():
    # Specify the path to the dataset
    combined_df = 'ml_api/classify-datasets/
combined_dataset.csv'
    data = pd.read_csv(combined_df)

    results = perform_eda(data, output_dir='ml_api/classify-
datasets')
    if results:
        print("EDA completed successfully.")
        print(results["summary_statistics"])
    else:
        print("EDA failed due to data loading issues.")

#
# ----------------------------------------------------------------------
# ----------------------

def class_level_analysis(data, output_dir='/Users/alicevang/
Desktop/696-proj/classify-project/front-end/src/data'):

    # First, we need to define a function that determines the
class level based on class_num
    def determine_level(class_num):
        if 100 <= class_num < 200:
            return '100-level'
        elif 200 <= class_num < 300:
            return '200-level'
        elif 300 <= class_num < 400:
            return '300-level'
        # Add more conditions as necessary
        else:
            return '400-level or higher'

    # Now, we apply this function to the class_num column to
create a new 'level' column
    data['level'] = data['class_num'].apply(determine_level)

    # Now let's retry the Class Level Analysis
    class_level_counts = data['level'].value_counts()
    plt.figure(figsize=(10, 6))
    sns.barplot(x=class_level_counts.index,
y=class_level_counts.values)
    plt.title('Distribution of Class Levels')
```

```
    plt.xlabel('Class Level')
    plt.ylabel('Number of Classes')
    plt.show()


    plot_path = os.path.join(output_dir,
'class_level_analysis.png')
    plt.savefig(plot_path)
    plt.close()

    return plot_path;

#
------------------------------------------------------------------
--------------------

#
------------------------------------------------------------------
--------------------

#
------------------------------------------------------------------
--------------------

def plot_distribution():
    # Load your combined dataset
    data = pd.read_csv(r'ml_api/classify-datasets/
combined_dataset.csv')

    # Day Distribution
    day_counts = data['day'].value_counts()

    # Start Time Distribution
    # Ensure the 'begin' column is in the correct format, then
get the count
    start_time_counts = data['begin'].value_counts()

    # Class Level Distribution
    # Assuming the class level can be inferred from class
numbers
    data['level'] = data['class_num'].apply(lambda x: '100-200'
if x < 300 else '300+')
    level_counts = data['level'].value_counts()

    # Professor Teaching Load
    teaching_load = data['instructor'].value_counts()
```

```
    # Day Distribution
    plt.figure(figsize=(12, 6))
    sns.barplot(x=day_counts.index, y=day_counts.values)
    plt.title('Class Distribution by Day')
    plt.xlabel('Day')
    plt.ylabel('Number of Classes')
    plt.show()

    # Start Time Distribution
    plt.figure(figsize=(14, 7))
    sns.barplot(x=start_time_counts.index,
y=start_time_counts.values)
    plt.title('Class Start Time Distribution')
    plt.xlabel('Start Time')
    plt.ylabel('Frequency')
    plt.xticks(rotation=90)
    plt.show()

    # Class Level Distribution
    plt.figure(figsize=(10, 6))
    sns.barplot(x=level_counts.index, y=level_counts.values)
    plt.title('Class Level Distribution')
    plt.xlabel('Class Level')
    plt.ylabel('Number of Classes')
    plt.show()

    # Professor Teaching Load
    plt.figure(figsize=(10, 8))
    sns.barplot(x=teaching_load.values, y=teaching_load.index)
    plt.title('Teaching Load by Instructor')
    plt.xlabel('Number of Classes')
    plt.ylabel('Instructor')
    plt.show()


#
------------------------------------------------------------------
----------------------

def conflict_analysis(spring_df, fall_df, combined_df):

    # Calculate summary statistics and compare
    # spring_stats = spring_df.describe()
    # fall_stats = fall_df.describe()

    # Comparison of summary statistics
    # comparison_stats = pd.concat([spring_stats, fall_stats],
```

```
axis=1, keys=['Spring', 'Fall'])

    # Identify peak times in both datasets -UNUSED-
    # spring_peak_times =
spring_df['begin'].value_counts().head()
    # fall_peak_times = fall_df['begin'].value_counts().head()

    # Define a function to calculate conflicts based on
overlapping times
    def calculate_conflicts(df):
        # Pivot table to count the number of classes at each
time slot for each day
        time_slots = pd.pivot_table(df, index='begin',
columns='day', aggfunc='size', fill_value=0)

        # A conflict is when there is more than one class at the
same time slot
        conflicts = time_slots[time_slots > 1].sum().sum()
        return conflicts

    # Calculate the number of conflicts for each term and the
combined dataset
    spring_conflicts = calculate_conflicts(spring_df)
    fall_conflicts = calculate_conflicts(fall_df)
    combined_conflicts = calculate_conflicts(combined_df)

    # return results
    results = {
            'spring': spring_conflicts,
            'fall': fall_conflicts,
            'combined': combined_conflicts
    }
    return results

#
------------------------------------------------------------------
--------------------

def create_heatmap(df, title):

    output_dir='/Users/alicevang/Desktop/696-proj/classify-
project/front-end/src/data'

     # Create a pivot table with counts of classes at each start
time and day
    heatmap_data = pd.pivot_table(df, index='begin',
columns='day', aggfunc='size', fill_value=0)
```

```
    # Any value above 1 indicates a conflict, so we'll highlight
those
    conflict_data = heatmap_data.where(heatmap_data > 1)

    plt.figure(figsize=(12, 8))
    sns.heatmap(conflict_data, cmap='Reds', linewidths=.5,
annot=True, fmt=".0f")
    plt.title(title)
    plt.xlabel('Day of the Week')
    plt.ylabel('Start Time')

    file_path = os.path.join(output_dir, f"{title.replace(' ',
'_').lower()}.png")
    plt.savefig(file_path)
    plt.close()  # Close the plot to free up memory
    return file_path

#
------------------------------------------------------------------
----------------------
def heatmap_conflicts(spring_df, fall_df, combined_df):

    # Create heatmaps for each dataset and collect the paths
    paths = []
    paths.append(create_heatmap(spring_df, 'Spring Term Class
Conflicts'))
    paths.append(create_heatmap(fall_df, 'Fall Term Class
Conflicts'))
    paths.append(create_heatmap(combined_df, 'Combined Term
Class Conflicts'))

    return paths

#
------------------------------------------------------------------
----------------------

# Improved function to convert 'begin' times to minutes since
midnight
'''
def parse_time_to_minutes(time_str):
    if pd.isnull(time_str) or isinstance(time_str, str) and
('TBA' in time_str or 'nan' in time_str):
        return None
    try:
        # Handle times without colon
```

```
        if time_str.isdigit() and len(time_str) == 4:
            time_str = time_str[:2] + ':' + time_str[2:]
        # Handle times with 'TBA' or similar issues
        if ':' not in time_str:
            return None
        hours, minutes = map(int, time_str.split(':'))
        return hours * 60 + minutes
    except (ValueError, TypeError) as e:
        print(f"Error converting time: {time_str} - {str(e)}")
        return None
    '''

  #
------------------------------------------------------------------
--------------------

def parse_time_to_minutes(time_str):
    if pd.isnull(time_str) or 'TBA' in time_str:
        return None
    try:
        # Split the time string into the time and AM/PM part
        time_part, period = time_str.strip().split()  #
Expecting format like "1:45 PM"
        hours, minutes = map(int, time_part.split(':'))

        # Convert hour to 24-hour format based on AM/PM
        if period.lower() == 'pm' and hours != 12:
            hours += 12
        elif period.lower() == 'am' and hours == 12:
            hours = 0  # Midnight is 0 hours in 24-hour time

        return hours * 60 + minutes
    except ValueError:
        return None

#
------------------------------------------------------------------
--------------------

# Function to calculate direct conflicts
def calculate_direct_conflicts(df):
    # Create a pivot table counting the number of classes at
each start time for each day
    time_counts = df.pivot_table(index='begin', columns='day',
aggfunc='size', fill_value=0)
    # Sum the counts where there are more than one class at the
same time slot
```

```
    conflicts = time_counts[time_counts > 1].sum().sum()
    return conflicts

#
------------------------------------------------------------------
--------------------

def output_conflict_score(spring_df, fall_df, combined_df):

    # Ensure 'begin' column is a string
    spring_df['begin'] = spring_df['begin'].astype(str)
    fall_df['begin'] = fall_df['begin'].astype(str)

    # Apply the time conversion to both datasets
    spring_df['start_time_minutes'] =
spring_df['begin'].apply(parse_time_to_minutes)
    fall_df['start_time_minutes'] =
fall_df['begin'].apply(parse_time_to_minutes)

    # Calculate conflict scores for each term and the combined
dataset
    spring_conflict_score =
calculate_direct_conflicts(spring_df)
    fall_conflict_score = calculate_direct_conflicts(fall_df)
    combined_conflict_score =
calculate_direct_conflicts(combined_df)

    results = {
        'Spring raw conflict score': spring_conflict_score,
        'Fall raw conflict score': fall_conflict_score,
        'combined raw conflict score': combined_conflict_score,
    }

    return results

#
------------------------------------------------------------------
--------------------

def process_prereq(combined_df):

    # Define nested functions
    def prerequisites_to_list(prereqs):
        if pd.isnull(prereqs) or prereqs == 'None':
            return []
        return prereqs.split()
```

```
    def validate_prereqs_format(prereqs_list):
        return isinstance(prereqs_list, list) and
all(isinstance(prereq, str) for prereq in prereqs_list)

    # Process prerequisites
    combined_df['prereq1_list'] =
combined_df['prereq1'].apply(prerequisites_to_list)
    combined_df['prereq2_list'] =
combined_df['prereq2'].apply(prerequisites_to_list)
    combined_df['all_prereqs'] = combined_df.apply(
        lambda row: row['prereq1_list'] + row['prereq2_list'],
axis=1
    )

    # Validate prerequisites
    combined_df['prereq1_valid'] =
combined_df['prereq1_list'].apply(validate_prereqs_format)
    combined_df['prereq2_valid'] =
combined_df['prereq2_list'].apply(validate_prereqs_format)
    combined_df['all_prereqs_valid'] =
combined_df['all_prereqs'].apply(validate_prereqs_format)

    # Extract invalid entries
    invalid_prereq1 = combined_df[~combined_df['prereq1_valid']]
    invalid_prereq2 = combined_df[~combined_df['prereq2_valid']]
    invalid_all_prereqs =
combined_df[~combined_df['all_prereqs_valid']]

    # Serialize data for JSON output
    results = {
        'invalid_prereq1':
invalid_prereq1.to_dict(orient='records'),
        'invalid_prereq2':
invalid_prereq2.to_dict(orient='records'),
        'invalid_all_prereqs':
invalid_all_prereqs.to_dict(orient='records')
    }

    return results

#
----------------------------------------------------------------
--------------------


#
----------------------------------------------------------------
```

```
--------------------

def prediction_pipeline(combined_df):
    combined_df = pd.read_csv(r'ml_api/classify-datasets/
combined_dataset.csv')

    file_name = 'predicted_schedule_complete.csv'
    output_dir = settings.MEDIA_ROOT
    file_path = os.path.join(output_dir, file_name)

    '''
    # Parse times to minutes
    def parse_time_to_minutes(time_str):
        if pd.isnull(time_str) or 'TBA' in time_str:
            return None
        try:
            hours, minutes = map(int, time_str.split(':'))
            return hours * 60 + minutes
        except ValueError:
            return None
    '''

    def parse_time_to_minutes(time_str):
        if pd.isnull(time_str) or 'TBA' in time_str:
            return None
        try:
            # Split the time string into the time and AM/PM part
            time_part, period = time_str.strip().split()  #
Expecting format like "1:45 PM"
            hours, minutes = map(int, time_part.split(':'))

            # Convert hour to 24-hour format based on AM/PM
            if period.lower() == 'pm' and hours != 12:
                hours += 12
            elif period.lower() == 'am' and hours == 12:
                hours = 0  # Midnight is 0 hours in 24-hour time

            return hours * 60 + minutes
        except ValueError:
            return None

    combined_df['start_time_minutes'] =
combined_df['begin'].apply(parse_time_to_minutes)
    #combined_df.dropna(subset=['start_time_minutes'],
inplace=True)  # Drop rows where time conversion failed

    # Define columns for preprocessing
```

```
    categorical_cols = ['day', 'instructor']
    numerical_cols = ['num', 'section', 'class_num', 'units']

    # Preprocessors
    numeric_transformer = Pipeline(steps=[
        ('imputer', SimpleImputer(strategy='median'))
    ])
    categorical_transformer = Pipeline(steps=[
        ('imputer', SimpleImputer(strategy='constant',
fill_value='missing')),
        ('onehot', OneHotEncoder(handle_unknown='ignore'))
    ])
    preprocessor = ColumnTransformer(transformers=[
        ('num', numeric_transformer, numerical_cols),
        ('cat', categorical_transformer, categorical_cols)
    ])

    # Model pipeline
    model_pipeline = Pipeline(steps=[
        ('preprocessor', preprocessor),
        ('regressor', DecisionTreeRegressor(random_state=42))
    ])

    # Split data
    X = combined_df.drop(['begin', 'end', 'start_time_minutes'],
axis=1)
    y = combined_df['start_time_minutes']
    X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

    # Fit model
    model_pipeline.fit(X_train, y_train)

    # Predict on both sets
    X_train['predicted_start_time'] =
model_pipeline.predict(X_train)
    X_test['predicted_start_time'] =
model_pipeline.predict(X_test)

    # Combine results
    full_results = pd.concat([X_train, X_test], axis=0)
    full_results['Term'] = 'Spring 2024'


    full_results.to_csv(file_path, index=False)

    return file_name
```

```
#
------------------------------------------------------------------
---------------------

def output_results(full_results):

    output_dir='/Users/alicevang/Desktop/696-proj/classify-
project/front-end/src/data'
    full_results = pd.read_csv(full_results)

    # Convert minutes back to time format
    full_results['predicted_start_time'] =
full_results['predicted_start_time'].apply(
        lambda minutes: f"{int(minutes // 60):02d}:{int(minutes
% 60):02d}"
    )

    # Calculate MSE for evaluation
    # mse = mean_squared_error(y_test,
X_test['predicted_start_time'])

    # Create a pivot table for heatmap data
    # Assuming 'time_slot' needs to be calculated from
'predicted_start_time'
    full_results['time_slot'] =
full_results['predicted_start_time'].apply(lambda x:
x.split(':')[0])  # Example transformation
    heatmap_data = full_results.pivot_table(index='day',
columns='time_slot', aggfunc='size', fill_value=0)


    # Generate the heatmap and save as PNG
    plt.figure(figsize=(12, 6))
    heatmap_data = heatmap_data.astype(int)
    sns.heatmap(heatmap_data, annot=True, fmt="d",
cmap="YlGnBu")
    plt.title("Class Distribution Heatmap")
    plt.ylabel('Day of Week')
    plt.xlabel('Time Slot')

    # Define the path for saving the heatmap
    heatmap_path = os.path.join(output_dir,
'schedule_heatmap.png')
    plt.savefig(heatmap_path)
    plt.close()
```

```
    # To calculate the number of conflicts, count overlapping
classes within each time slot.
    conflicts = heatmap_data.applymap(lambda x: x - 1 if x > 1
else 0).sum().sum()

    # Prepare the data for JSON response
    response_data = {
        # "mse": mse,
        "conflicts": int(conflicts),
        "heatmap_path": heatmap_path,
    }

    return response_data

#
------------------------------------------------------------------
----------------------


#
------------------------------------------------------------------
----------------------

#
------------------------------------------------------------------
----------------------
```