Streamlit

import streamlit as st

st.title("✅ Streamlit is working!")

st.write("Hello from your PyCharm + Streamlit setup.")

st.write("Test")

st.text\_input("Interactive dashboard for Education Policy Maker ")

Descriptive

import streamlit as st

import pandas as pd

import plotly.express as px

import matplotlib.pyplot as plt # Added for the Overview section's matplotlib plot

import seaborn as sns # Added for the Overview section's seaborn plot

import json # Added for the Overview section's geojson

import unicodedata # Added for the Overview section's normalization

import io # Added for the Overview section's export

# === CONFIG ===

st.set\_page\_config(page\_title="📊 Bogotá Descriptive Dashboard", layout="wide")

# === LOAD DATA ===

FILE\_PATH = "https://github.com/Vinothkumar-Base/concession-school-analysis/raw/refs/heads/main/MASTER\_CLEANED1.xlsx”

df = pd.read\_excel(FILE\_PATH, dtype={"Dane Code": str})

# This line is crucial: it makes "Name of Establishment" -> "Name Of Establishment"

df.columns = df.columns.str.strip().str.title()

# === RENAME INCONSISTENT COLUMNS (for main df) ===

# We are \*not\* renaming "Name Of Establishment" here, as per your request.

df.rename(columns={

"Enrolled Students (Last 3 Years)": "Enrollment",

"Pass Rate": "Pass Rate",

"Dropout Rate": "Dropout Rate",

"Failure Rate": "Failure Rate",

"Absorption Rate In Higher Education": "Absorption Rate"

}, inplace=True)

# === FILTER ONLY CONCESSION SCHOOLS ===

df["Concession"] = df["Concession"].astype(str).str.strip().str.lower()

df = df[df["Concession"] == "yes"]

# Keep "Name Of Establishment" here

df = df.dropna(subset=["Administrator", "Name Of Establishment", "Year"])

df["Administrator"] = df["Administrator"].str.upper()

df["Year"] = pd.to\_numeric(df["Year"], errors="coerce").astype(int)

# === CLEAN NUMERIC COLUMNS ===

subject\_cols = [

"Mathematics Index", "Natural Sciences Index", "Social And Citizenship Index",

"Critical Reading Index", "English Index", "Total Index"

]

rate\_cols = ["Pass Rate", "Dropout Rate", "Failure Rate", "Absorption Rate"]

enroll\_cols = ["Evaluated (Last 3 Years)"] # Enrollment removed

rank\_col = ["Ranking"]

all\_indicators = subject\_cols + rate\_cols + enroll\_cols

for col in all\_indicators:

if col in df.columns:

df[col] = pd.to\_numeric(df[col], errors="coerce")

# === SIDEBAR SECTION SELECTOR ===

with st.sidebar:

st.header("Select Section")

section = st.selectbox("Choose Section", ["Overview", "By Locality", "By Concession Schools"])

# === DYNAMIC TITLE ===

section\_titles = {

"Overview": "📊 Overview of Bogotá Education System",

"By Locality": "📍 Analysis by Locality",

"By Concession Schools": "🏫 Concession School Performance"

}

st.title(section\_titles.get(section, "📊 Education Dashboard"))

# === SECTION: BY CONCESSION SCHOOLS ===

if section == "By Concession Schools":

with st.sidebar:

st.header("🔎 Filters")

view\_mode = st.radio("Select View Mode", ["Total View", "Administrator Level", "School Level"])

indicator = st.selectbox("Select Indicator", all\_indicators)

admin\_filter = []

school\_filter = []

if view\_mode in ["Administrator Level", "School Level"]:

admin\_filter = st.multiselect("Select Administrator(s)", sorted(df["Administrator"].unique()))

if view\_mode == "School Level" and admin\_filter:

# Keep "Name Of Establishment" here

filtered\_schools = sorted(df[df["Administrator"].isin(admin\_filter)]["Name Of Establishment"].unique())

school\_filter = st.multiselect("Select School(s)", filtered\_schools)

# === TOTAL VIEW ===

if view\_mode == "Total View":

st.subheader(f"📊 Average {indicator} per Administrator (All Years)")

avg\_df = df.groupby("Administrator")[indicator].mean().reset\_index().sort\_values(by=indicator, ascending=False)

fig\_avg = px.bar(avg\_df, x="Administrator", y=indicator, color="Administrator",

title=f"Average {indicator} per Administrator (All Years)", height=700)

fig\_avg.update\_layout(yaxis=dict(tickfont=dict(size=13)), xaxis=dict(tickfont=dict(size=13)))

st.plotly\_chart(fig\_avg, use\_container\_width=True)

st.subheader(f"📈 {indicator} Over Time by Administrator")

chart\_data = df.groupby(["Administrator", "Year"])[indicator].mean().reset\_index()

fig = px.line(chart\_data, x="Year", y=indicator, color="Administrator",

markers=True, title=f"{indicator} Over Time by Administrator", height=600)

fig.update\_layout(xaxis=dict(dtick=1))

st.plotly\_chart(fig, use\_container\_width=True)

st.subheader("🏅 Ranking Contribution per Administrator")

rank\_df = df.dropna(subset=["Ranking"])

rank\_summary = rank\_df.groupby(["Administrator", "Ranking"]).size().reset\_index(name="Count")

total\_counts = rank\_summary.groupby("Administrator")["Count"].transform("sum")

rank\_summary["Percentage"] = (rank\_summary["Count"] / total\_counts \* 100).round(2)

fig\_rank = px.bar(

rank\_summary, x="Administrator", y="Count", color="Ranking", text="Percentage",

title="Ranking Distribution by Administrator (%)", height=600

)

fig\_rank.update\_traces(textposition='inside')

fig\_rank.update\_layout(barmode='stack', xaxis={'categoryorder': 'total descending'})

st.plotly\_chart(fig\_rank, use\_container\_width=True)

# === ADMINISTRATOR LEVEL ===

elif view\_mode == "Administrator Level" and admin\_filter:

st.subheader(f"📈 {indicator} Over Time by Administrator")

chart\_data = df[df["Administrator"].isin(admin\_filter)]

chart\_grouped = chart\_data.groupby(["Administrator", "Year"])[indicator].mean().reset\_index()

fig = px.line(chart\_grouped, x="Year", y=indicator, color="Administrator",

markers=True, title=f"{indicator} Over Time by Administrator", height=600)

fig.update\_layout(xaxis=dict(dtick=1))

st.plotly\_chart(fig, use\_container\_width=True)

st.subheader("📊 Ranking Breakdown by School")

rank\_school = chart\_data.dropna(subset=["Ranking"])

# Keep "Name Of Establishment" here

fig\_rank\_school = px.histogram(rank\_school, x="Ranking", color="Name Of Establishment", barmode="group",

title="Ranking Composition by School", height=500)

st.plotly\_chart(fig\_rank\_school, use\_container\_width=True)

st.markdown("### 🏫 Schools under Selected Administrator(s)")

# Keep "Name Of Establishment" here

school\_list = chart\_data[["Administrator", "Name Of Establishment"]].drop\_duplicates().sort\_values(["Administrator", "Name Of Establishment"])

st.dataframe(school\_list, use\_container\_width=True)

# === SCHOOL LEVEL ===

elif view\_mode == "School Level" and school\_filter:

for school in school\_filter:

# Keep "Name Of Establishment" here

school\_data = df[df["Name Of Establishment"] == school]

st.subheader(f"📚 {school} — {indicator} Over Time")

fig\_school = px.line(school\_data, x="Year", y=indicator, color="Administrator",

markers=True, title=f"{indicator} Over Time at {school}", height=500)

fig\_school.update\_layout(xaxis=dict(dtick=1))

st.plotly\_chart(fig\_school, use\_container\_width=True)

st.markdown("#### 🔍 Key Details")

recent\_data = school\_data.sort\_values("Year", ascending=False).head(1).T

recent\_data.columns = ["Most Recent Record"]

st.dataframe(recent\_data, use\_container\_width=True)

# === SECTION: OVERVIEW ===

elif section == "Overview":

with st.sidebar:

st.header("📊 Overview Filters")

# ============================== #

# 📁 Load Datasets

# ============================== #

school\_path = r"C:\Study document\6611 Dataset\Final\Copy of bogota\_all\_schools\_ranking\_saber\_11.xlsx"

map\_school\_path = r"C:\Study document\6611 Dataset\Final\Copy of bogota\_all\_schools\_ranking\_saber\_11.xlsx"

poverty\_path = r"C:\Study document\6611 Dataset\Final\poverty\_inequality\_data\_bogota.xlsx"

geojson\_path = r"C:\Study document\6611 Dataset\Final\bta\_localidades.geojson"

school\_df = pd.read\_excel(school\_path)

# Apply .str.title() immediately for consistency

school\_df.columns = school\_df.columns.str.strip().str.title()

map\_school\_df = pd.read\_excel(map\_school\_path)

# Apply .str.title() immediately for consistency

map\_school\_df.columns = map\_school\_df.columns.str.strip().str.title()

poverty\_df = pd.read\_excel(poverty\_path)

with open(geojson\_path, "r", encoding="utf-8") as f:

bogota\_geo = json.load(f)

# ============================== #

# 🔤 Normalize Location Names

# ============================== #

def normalize\_location(s):

s = str(s)

s = unicodedata.normalize('NFKD', s).encode('ASCII', 'ignore').decode('utf-8')

return s.strip().upper()

location\_name\_map = {

"SANTAFE": "SANTA FE",

"LA CANDELARIA": "CANDELARIA",

"ANTONIO NARINO": "ANTONIO NARIÑO"

}

# ============================== #

# 🧹 Clean and Prepare Data

# ============================== #

poverty\_df = poverty\_df.rename(columns={

"Año": "year",

"Localidad": "location\_name",

"Indicador": "indicator",

"Valor": "value"

})

poverty\_df["location\_name"] = poverty\_df["location\_name"].apply(normalize\_location)

poverty\_df["location\_name"] = poverty\_df["location\_name"].replace(location\_name\_map)

poverty\_df["value"] = pd.to\_numeric(poverty\_df["value"], errors="coerce")

school\_df = school\_df.rename(columns={

"Year": "year",

"Location Name": "location\_name", # After .title(), "Location name" becomes "Location Name"

"Sector": "sector",

"Concession": "concession",

"Mathematics Index": "mathematics\_index",

"Natural Sciences Index": "natural\_sciences\_index",

"Social And Citizenship Index": "social\_and\_citizenship\_index", # After .title(), "Social and Citizenship Index" becomes "Social And Citizenship Index"

"Critical Reading Index": "critical\_reading\_index",

"English Index": "english\_index",

"Total Index": "total\_index",

"Ranking": "ranking"

# Removed "Name Of Establishment": "school\_name" here as per your request

})

school\_df = school\_df.dropna(subset=['location\_name'])

school\_df = school\_df[school\_df['location\_name'].str.strip() != ""]

school\_df['location\_name'] = school\_df['location\_name'].apply(normalize\_location)

school\_df['location\_name'] = school\_df['location\_name'].replace(location\_name\_map)

map\_school\_df = map\_school\_df.rename(columns={

"Location Name": "location\_name", # After .title(), "Location name" becomes "Location Name"

# Removed "Name Of Establishment": "school\_name" here as per your request

})

map\_school\_df['location\_name'] = map\_school\_df['location\_name'].apply(normalize\_location)

map\_school\_df['location\_name'] = map\_school\_df['location\_name'].replace(location\_name\_map)

school\_df["sector\_type"] = school\_df.apply(

lambda row: "CONCESSION" if str(row["sector"]).upper() == "PUBLIC" and str(row["concession"]).strip().upper() == "YES"

else ("PUBLIC" if str(row["sector"]).upper() == "PUBLIC" else "PRIVATE"),

axis=1

)

# ============================== #

# 🧭 Sidebar Filters

# ============================== #

school\_years = sorted(school\_df["year"].dropna().unique())

selected\_year = st.sidebar.selectbox("Select Year", school\_years, index=len(school\_years) - 1)

localities = ['ALL'] + sorted(school\_df['location\_name'].unique())

selected\_locality = st.sidebar.selectbox("Select Locality", localities)

poverty\_valid\_years = [2011, 2014, 2017, 2021]

st.sidebar.markdown("---")

st.sidebar.markdown(f"🗓️ School data: \*\*{min(school\_years)}–{max(school\_years)}\*\*\n\n📉 Poverty data: \*\*{', '.join(map(str, poverty\_valid\_years))}\*\*")

# ============================== #

# 🎯 Filter Data

# ============================== #

filtered\_df = school\_df[school\_df["year"] == selected\_year].copy()

# Ensure "Year" in map\_school\_df is handled correctly if it's not already numeric

if 'Year' in map\_school\_df.columns:

map\_school\_df['Year'] = pd.to\_numeric(map\_school\_df['Year'], errors='coerce').fillna(-1).astype(int) # Handle potential NaNs for conversion

map\_filtered\_df = map\_school\_df[map\_school\_df["Year"] == selected\_year].copy()

if selected\_locality != 'ALL':

filtered\_df = filtered\_df[filtered\_df['location\_name'] == selected\_locality]

map\_filtered\_df = map\_filtered\_df[map\_filtered\_df['location\_name'] == selected\_locality]

poverty\_years\_available = sorted(poverty\_df['year'].unique())

latest\_poverty\_year = max([y for y in poverty\_years\_available if y <= selected\_year], default=None)

poverty\_year = poverty\_df[(poverty\_df["year"] == latest\_poverty\_year) & (poverty\_df["indicator"].str.lower().str.contains("extrema"))]

if selected\_locality != 'ALL':

poverty\_year = poverty\_year[poverty\_year['location\_name'] == selected\_locality]

# ============================== #

# 📥 Export Data

# ============================== #

st.markdown("## 📥 Export Data")

csv = filtered\_df.to\_csv(index=False).encode('utf-8')

st.download\_button("⬇️ Download Filtered Data (CSV)", data=csv, file\_name=f"filtered\_schools\_{selected\_year}.csv", mime='text/csv')

excel\_buffer = io.BytesIO()

with pd.ExcelWriter(excel\_buffer, engine='xlsxwriter') as writer:

filtered\_df.to\_excel(writer, index=False, sheet\_name="Filtered Data")

st.download\_button("📊 Download Filtered Data (Excel)", data=excel\_buffer.getvalue(), file\_name=f"filtered\_schools\_{selected\_year}.xlsx", mime='application/vnd.openxmlformats-officedocument.spreadsheetml.sheet')

# ============================== #

# 📊 KPI Cards

# ============================== #

st.title(":bar\_chart: Bogotá Educational Index")

total = len(filtered\_df)

private = len(filtered\_df[filtered\_df["sector\_type"] == "PRIVATE"])

public = len(filtered\_df[filtered\_df["sector\_type"] == "PUBLIC"])

concession = len(filtered\_df[filtered\_df["sector\_type"] == "CONCESSION"])

col1, col2, col3, col4 = st.columns(4)

col1.metric("🎓 Total Schools", total)

col2.metric("🏢 Private", private)

col3.metric("🏫 Public", public)

col4.metric("🪡 Concession (PPP)", concession)

# ============================== #

# ============================== #

# 🌍 Maps

# ============================== #

col\_map1, col\_map2 = st.columns(2)

with col\_map1:

st.subheader("School Distribution")

# Ensure 'Name Of Establishment' exists in map\_filtered\_df if it's the identifier for counting

# Assuming you want to count schools, not necessarily display their name here,

# but if `map\_school\_df` has `Name Of Establishment`, it's good.

# It's better to group by location\_name as done before.

map\_df = map\_filtered\_df.groupby("location\_name").size().reset\_index(name="count")

fig = px.choropleth\_mapbox(

map\_df, geojson=bogota\_geo, locations="location\_name",

featureidkey="properties.NOMBRE", color="count",

color\_continuous\_scale="Viridis", mapbox\_style="carto-positron",

zoom=9.2, center={"lat": 4.65, "lon": -74.1}, opacity=0.85, height=500

)

st.plotly\_chart(fig, use\_container\_width=True)

with col\_map2:

st.subheader("Extreme Monetary Poverty")

if not poverty\_year.empty:

fig\_poverty = px.choropleth\_mapbox(

poverty\_year, geojson=bogota\_geo, locations="location\_name",

featureidkey="properties.NOMBRE", color="value",

color\_continuous\_scale="Reds", mapbox\_style="carto-positron",

zoom=9.2, center={"lat": 4.65, "lon": -74.1}, opacity=0.85, height=500

)

st.plotly\_chart(fig\_poverty, use\_container\_width=True)

else:

st.info("Poverty data not available for this year.")

# ============================== #

# 📈 Additional Charts

# ============================== #

st.subheader("Poverty vs School Availability")

# This `school\_count` is from map\_filtered\_df, which should have 'Name Of Establishment'

school\_counts = map\_filtered\_df.groupby("location\_name").size().reset\_index(name="school\_count")

poverty\_scores = poverty\_year[["location\_name", "value"]].rename(columns={"value": "extreme\_poverty"})

merged = pd.merge(school\_counts, poverty\_scores, on="location\_name", how="inner")

fig\_scatter = px.scatter(

merged, x="extreme\_poverty", y="school\_count", text="location\_name",

size="school\_count", color="extreme\_poverty", color\_continuous\_scale="Plasma"

)

fig\_scatter.update\_traces(textposition="top center")

st.plotly\_chart(fig\_scatter)

st.subheader("Sector Distribution by Locality")

sector\_data = filtered\_df.groupby(["location\_name", "sector\_type"]).size().reset\_index(name="count")

fig\_sector = px.bar(sector\_data, x="location\_name", y="count", color="sector\_type", barmode="stack", height=500)

st.plotly\_chart(fig\_sector)

st.subheader("Saber 11 Index by Sector over Time")

subject\_cols = [

"mathematics\_index", "natural\_sciences\_index",

"social\_and\_citizenship\_index", "critical\_reading\_index",

"english\_index", "total\_index"

]

selected\_subject = st.selectbox("Select Subject", subject\_cols, index=subject\_cols.index("total\_index"))

avg\_by\_sector = school\_df.groupby(["year", "sector\_type"])[selected\_subject].mean().reset\_index()

fig = px.line(avg\_by\_sector, x="year", y=selected\_subject, color="sector\_type", markers=True)

fig.update\_layout(yaxis\_title="Saber 11 Index", height=500)

st.plotly\_chart(fig)

st.subheader("Ranking Distribution by Sector")

rank\_data = filtered\_df.dropna(subset=['ranking'])

rank\_grouped = rank\_data.groupby(['sector\_type', 'ranking']).size().reset\_index(name='count')

fig\_rank = px.sunburst(rank\_grouped, path=['sector\_type', 'ranking'], values='count')

st.plotly\_chart(fig\_rank)

st.subheader("Poverty vs Saber 11 Average")

poverty\_rate = poverty\_year[["location\_name", "value"]].rename(columns={"value": "poverty\_rate"})

school\_year = filtered\_df.copy()

index\_cols = [col for col in school\_year.columns if col.endswith("\_index") and col != "total\_index"]

school\_year["saber\_avg"] = school\_year[index\_cols].mean(axis=1)

saber\_avg = school\_year.groupby("location\_name")["saber\_avg"].mean().reset\_index()

merged = pd.merge(poverty\_rate, saber\_avg, on="location\_name", how="inner")

merged = merged.sort\_values(by="poverty\_rate", ascending=False)

fig, ax1 = plt.subplots(figsize=(16, 7))

sns.barplot(x="location\_name", y="poverty\_rate", data=merged, ax=ax1, color="orange", zorder=1)

ax2 = ax1.twinx()

sns.lineplot(x="location\_name", y="saber\_avg", data=merged, ax=ax2, color="green", marker="o", linewidth=2.5, zorder=2)

ax1.set\_ylabel("Extreme Poverty Rate (%)")

ax2.set\_ylabel("Saber 11 Avg Index")

ax1.set\_xticklabels(merged["location\_name"], rotation=45, ha="right")

fig.tight\_layout()

st.pyplot(fig)

# 🧠 Average, Top 10, Bottom 10 Saber Index

# ============================== #

st.subheader("📌 Saber 11 Performance Summary")

index\_df = filtered\_df.dropna(subset=["total\_index"])

if not index\_df.empty:

avg\_score = index\_df["total\_index"].mean()

max\_score = index\_df["total\_index"].max()

min\_score = index\_df["total\_index"].min()

st.markdown(f"""

- 🔢 \*\*Average Total Index\*\*: {avg\_score:.2f}

- 🏆 \*\*Highest Total Index\*\*: {max\_score:.2f}

- 🚨 \*\*Lowest Total Index\*\*: {min\_score:.2f}

""")

st.markdown("### 🥇 Top 10 Schools by Total Index")

top10 = index\_df.sort\_values(by="total\_index", ascending=False).head(10)[

["Name Of Establishment", "location\_name", "total\_index", "sector\_type"]] # KEPT "Name Of Establishment"

st.dataframe(top10.reset\_index(drop=True))

st.markdown("### 🛑 Bottom 10 Schools by Total Index")

bottom10 = index\_df.sort\_values(by="total\_index", ascending=True).head(10)[

["Name Of Establishment", "location\_name", "total\_index", "sector\_type"]] # KEPT "Name Of Establishment"

st.dataframe(bottom10.reset\_index(drop=True))

else:

st.info("No index data available for this year and locality.")

# === SECTION: LOCALITY PLACEHOLDER ===

elif section == "By Locality":

st.info("📍 Locality-based analysis is not yet available.")

Predictive

import streamlit as st

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.impute import SimpleImputer

from sklearn.pipeline import Pipeline

from sklearn.preprocessing import StandardScaler, OneHotEncoder, LabelEncoder

from sklearn.compose import ColumnTransformer

from sklearn.ensemble import RandomForestClassifier

from sklearn.model\_selection import train\_test\_split

from sklearn.cluster import KMeans

from sklearn.decomposition import PCA

from statsmodels.tsa.api import SimpleExpSmoothing, Holt

from statsmodels.tsa.arima.model import ARIMA

from statsmodels.tsa.stattools import adfuller

import plotly.express as px

import plotly.graph\_objects as go

from xgboost import XGBClassifier, plot\_importance # Import necessary XGBoost components

import xgboost as xgb # For xgb\_model.get\_booster()

import warnings

import os

warnings.filterwarnings("ignore")

# === Sidebar Navigation ===

st.set\_page\_config(page\_title="Bogotá School Analytics", layout="wide")

section = st.sidebar.selectbox(

"Select Section",

["1. School Classification", "2. Time Series Prediction", "3. School Ranking Prediction"]

)

# ---

# ## Section 1: School Classification

# ---

if section == "1. School Classification":

st.title("🎯 School Ranking Classifier")

st.info("This section classifies schools into rankings (A+, A, B, C, D) using a Random Forest model.")

# === Load and clean data ===

file\_path = r"C:\Study document\6611 Dataset\Final\MASTER\_CLEANED1.xlsx"

df = pd.read\_excel(file\_path)

# Apply standardization immediately after loading

df.columns = df.columns.str.strip().str.title()

# Filter and clean

df = df[df['Ranking'].isin(['A+', 'A', 'B', 'C', 'D'])].dropna(subset=['Ranking'])

df['Sector'] = df['Sector'].str.upper().str.strip()

df['Sector'] = df['Sector'].apply(lambda x: 'PRIVATE' if 'PRIVATE' in x else 'PUBLIC')

# Use standardized column name

if 'Location Name' in df.columns:

df['Location Name'] = df['Location Name'].astype(str).str.upper().str.strip()

else:

st.error("Missing 'Location Name' column in Section 1. Please check your Excel file.")

st.stop()

# Define features

categorical\_cols = ['Concession', 'Sector', 'Location Name']

numerical\_cols = [

"Enrolled Students (Last 3 Years)",

"Mathematics Index", "Natural Sciences Index", "Social And Citizenship Index",

"Critical Reading Index", "English Index", "Total Index"

]

df = df.dropna(subset=numerical\_cols)

# Encode target label

label\_encoder = LabelEncoder()

df['Ranking\_Label'] = label\_encoder.fit\_transform(df['Ranking'])

# Preprocessing

num\_pipeline = Pipeline([

("imputer", SimpleImputer(strategy="mean")),

("scaler", StandardScaler())

])

cat\_pipeline = Pipeline([

("imputer", SimpleImputer(strategy="most\_frequent")),

("encoder", OneHotEncoder(handle\_unknown="ignore"))

])

preprocessor = ColumnTransformer([

("num", num\_pipeline, numerical\_cols),

("cat", cat\_pipeline, categorical\_cols)

])

# Train/Test split

X = df[categorical\_cols + numerical\_cols]

y = df["Ranking\_Label"]

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, stratify=y, random\_state=42)

# Build model pipeline

pipeline = Pipeline([

("preprocessor", preprocessor),

("classifier", RandomForestClassifier(n\_estimators=100, random\_state=42))

])

pipeline.fit(X\_train, y\_train)

# === Sidebar Input Form ===

st.sidebar.header("📥 Enter New School Data")

with st.sidebar.form("input\_form"):

user\_input = {

"Concession": st.selectbox("Concession", df["Concession"].dropna().unique()),

"Sector": st.selectbox("Sector", df["Sector"].dropna().unique()),

"Location Name": st.selectbox("Location", df["Location Name"].dropna().unique()), # Use standardized name

"Enrolled Students (Last 3 Years)": st.number\_input("Enrolled students", value=100),

"Mathematics Index": st.slider("Math Index", 0.0, 1.0, 0.5),

"Natural Sciences Index": st.slider("Science Index", 0.0, 1.0, 0.5),

"Social And Citizenship Index": st.slider("Social Index", 0.0, 1.0, 0.5),

"Critical Reading Index": st.slider("Reading Index", 0.0, 1.0, 0.5),

"English Index": st.slider("English Index", 0.0, 1.0, 0.5),

"Total Index": st.slider("Total Index", 0.0, 1.0, 0.5)

}

submitted = st.form\_submit\_button("🔮 Predict Ranking")

# === Output Prediction and Feature Importance ===

if submitted:

user\_df = pd.DataFrame([user\_input])

pred = pipeline.predict(user\_df)[0]

pred\_label = label\_encoder.inverse\_transform([pred])[0]

st.success(f"🎯 Predicted School Ranking: \*\*{pred\_label}\*\*")

raw\_feature\_names = pipeline.named\_steps['preprocessor'].get\_feature\_names\_out()

def clean\_feature\_name(raw):

if raw.startswith("num\_\_"):

return raw.replace("num\_\_", "").replace("\_", " ").title()

elif raw.startswith("cat\_\_"):

if "Sector\_" in raw:

return "Sector"

elif "Location Name\_" in raw: # Use standardized name

return "Location"

elif "Concession\_" in raw:

return "Concession"

else:

return "Category"

return raw.title()

cleaned\_features = [clean\_feature\_name(f) for f in raw\_feature\_names]

model = pipeline.named\_steps['classifier']

importances = model.feature\_importances\_

importance\_df = pd.DataFrame({

"Feature": cleaned\_features,

"Importance": importances

})

importance\_df = importance\_df.groupby("Feature").sum().reset\_index()

importance\_df = importance\_df.sort\_values("Importance", ascending=False).head(10)

fig = px.bar(importance\_df, x="Importance", y="Feature", orientation="h",

title="🔍 Top 10 Feature Importances", color="Importance",

color\_continuous\_scale="Cividis")

fig.update\_layout(height=420)

st.plotly\_chart(fig, use\_container\_width=True)

# ---

# ## Section 2: Time Series Prediction

# ---

elif section == "2. Time Series Prediction":

st.title("📈 School Indicator Time Series Predictor")

DATA\_FILE = r"C:\Users\linht\PycharmProjects\pythonProject3\MASTER\_CLEANED1.xlsx" # Adjusted path based on typical project structure

PREDICTION\_YEARS = [2025, 2026, 2027]

INDICATORS = [

'Total Index', 'Pass Rate', 'Dropout Rate', # Use standardized names

'Failure Rate', 'Enrollment', 'Absorption Rate' # Use standardized names

]

@st.cache\_data

def load\_data():

if not os.path.exists(DATA\_FILE):

st.error(f"File not found: {DATA\_FILE}")

st.stop()

df = pd.read\_excel(DATA\_FILE, dtype={'Dane Code': str})

# Standardize column names immediately after loading

df.columns = df.columns.str.strip().str.title()

# Now, rename using the \*standardized\* source column names

df.rename(columns={

'Enrolled Students (Last 3 Years)': 'Enrollment', # This should now exist after .title()

'Absorption Rate In Higher Education': 'Absorption Rate', # This should now exist after .title()

}, inplace=True)

if 'Year' in df.columns:

df['Year'] = pd.to\_numeric(df['Year'], errors='coerce').fillna(-1).astype(int)

df = df[df['Year'] != -1]

else:

st.error("Missing 'Year' column in the dataset. Cannot proceed.")

st.stop()

for col in INDICATORS:

if col in df.columns: # Check if column exists after standardization and renames

df[col] = pd.to\_numeric(

df[col].astype(str).str.replace(',', '.', regex=False).str.replace('%', '', regex=False),

errors='coerce'

)

if col in ['Pass Rate', 'Dropout Rate', 'Failure Rate', 'Absorption Rate']:

df[col] /= 100

else:

st.warning(

f"Indicator column '{col}' not found in the dataset. It will not be available for prediction.")

df['Dane Code'] = df['Dane Code'].astype(str).str.strip().str.replace(r'\.0$', '', regex=True)

df.dropna(subset=['Dane Code'], inplace=True)

return df

def predict\_time\_series(series, future\_years, indicator, dane):

s = series.dropna()

n = len(s)

if n == 0:

return pd.Series(np.nan, index=future\_years)

if n < 3: # Need at least 3 points for meaningful forecasting, otherwise just extend last value

return pd.Series(s.iloc[-1], index=future\_years)

try:

model = ARIMA(s, order=(1, 1, 1))

fit = model.fit()

fc = fit.get\_forecast(steps=len(future\_years)).predicted\_mean

if fc.isnull().any() or np.isinf(fc).any():

raise ValueError("NaN or Inf in ARIMA forecast.")

return pd.Series(fc.values, index=future\_years)

except Exception as e:

# st.warning(f"ARIMA failed for {dane} {indicator}: {e}") # Optional: uncomment for detailed debug

pass

try:

fit = Holt(s, initialization\_method="estimated").fit()

fc = fit.forecast(len(future\_years))

if fc.isnull().any() or np.isinf(fc).any():

raise ValueError("NaN or Inf in Holt forecast.")

return pd.Series(fc.values, index=future\_years)

except Exception as e:

# st.warning(f"Holt failed for {dane} {indicator}: {e}") # Optional: uncomment for detailed debug

pass

try:

fit = SimpleExpSmoothing(s, initialization\_method="estimated").fit()

fc = fit.forecast(len(future\_years))

if fc.isnull().any() or np.isinf(fc).any():

raise ValueError("NaN or Inf in SES forecast.")

return pd.Series(fc.values, index=future\_years)

except Exception as e:

# st.warning(f"SES failed for {dane} {indicator}: {e}") # Optional: uncomment for detailed debug

pass

# Fallback: if all models fail or data is insufficient, predict the last value

return pd.Series(s.iloc[-1], index=future\_years)

df = load\_data() # Load the data with the corrected column titles

dane\_input = st.text\_input("Enter DANE Code:").strip()

if dane\_input:

school = df[df['Dane Code'] == dane\_input].sort\_values('Year')

if school.empty:

st.warning("No data for that DANE Code.")

else:

# Use the standardized column name "Name Of Establishment"

if "Name Of Establishment" in school.columns:

name = school["Name Of Establishment"].iloc[0]

else:

name = f"School with DANE Code: {dane\_input}" # Fallback name

st.warning("'Name Of Establishment' column not found, using DANE Code as name.")

st.subheader(f"{name} (DANE Code: {dane\_input})")

min\_y, max\_y = school['Year'].min(), school['Year'].max()

years = list(range(min\_y, max\_y + 1)) + PREDICTION\_YEARS

results = pd.DataFrame(index=years)

for ind in INDICATORS:

if ind in school.columns: # Ensure indicator column exists after load and renames

hist = school.set\_index('Year')[ind].reindex(range(min\_y, max\_y + 1))

fc = predict\_time\_series(hist, PREDICTION\_YEARS, ind, dane\_input)

combined = pd.concat([hist, fc])

combined.index = combined.index.astype(int)

results[ind] = combined

else:

st.warning(

f"Indicator '{ind}' not found in data for DANE Code {dane\_input}. Skipping prediction for this indicator.")

# Filter out indicators not found in the DataFrame from the dropdown list

available\_indicators = [ind for ind in INDICATORS if ind in results.columns]

if available\_indicators:

choose = st.selectbox("Choose an indicator:", available\_indicators)

plot\_df = results[choose].reset\_index().rename(columns={'index': 'Year', choose: 'Value'})

plot\_df['Type'] = plot\_df['Year'].apply(

lambda y: 'Prediction' if y in PREDICTION\_YEARS else 'Historical')

plot\_df.dropna(subset=['Value'], inplace=True)

if plot\_df.empty:

st.warning("No data to plot for the selected indicator.")

else:

fig = px.line(plot\_df, x='Year', y='Value', line\_dash='Type',

title=f"{choose} trend for {name}")

fig.update\_xaxes(dtick=1)

st.plotly\_chart(fig, use\_container\_width=True)

else:

st.warning("No valid indicators found for plotting.")

# ---

# ## Section 3: School Ranking Prediction

# ---

elif section == "3. School Ranking Prediction":

st.title("📊 School Ranking Analysis & Projection")

# --- File uploader and default example ---

st.sidebar.header("📂 Upload Dataset")

user\_file = st.sidebar.file\_uploader("Upload Excel File", type=["xlsx"])

# --- Load default example ---

@st.cache\_data # Added cache for this function

def load\_data\_ranking():

path = r"C:\Study document\6611 Dataset\Final\MASTER\_CLEANED1.xlsx" # Adjusted path

df\_ranking = pd.read\_excel(path)

# Standardize column names here too

df\_ranking.columns = df\_ranking.columns.str.strip().str.title()

return df\_ranking

if user\_file:

df = pd.read\_excel(user\_file)

df.columns = df.columns.str.strip().str.title() # Apply here if user uploads

else:

df = load\_data\_ranking() # Use the consistent loader

st.info("Using example dataset. Upload your own file to analyze custom data.")

# --- Preprocessing ---

# Ensure 'Total Index' exists before dropping NaNs

if 'Total Index' in df.columns:

df = df.dropna(subset=['Total Index'])

else:

st.error("Missing 'Total Index' column, cannot proceed with ranking prediction.")

st.stop()

# Handle 'Year' column consistently

# Check for 'Evaluated (Last 3 Years)' (standardized)

if 'Evaluated (Last 3 Years)' in df.columns:

if 'Year' not in df.columns:

df['Year'] = pd.to\_datetime(df['Evaluated (Last 3 Years)'], errors='coerce').dt.year.fillna(2024).astype(

int)

else:

df['Year'] = df['Year'].astype(int)

elif 'Year' in df.columns:

df['Year'] = df['Year'].astype(int)

else:

st.warning("Missing 'Year' or 'Evaluated (Last 3 Years)' column. Defaulting Year to 2024.")

df['Year'] = 2024 # Default year if no date column found

# Columns to standardize and fillna (ensure these are the standardized names)

cols\_to\_process = ['Concession', 'Ranking', 'Location Name']

for col in cols\_to\_process:

if col in df.columns:

df[col] = df[col].astype(str).str.strip()

df[col] = df[col].fillna("Unknown")

else:

st.warning(f"Column '{col}' not found for processing. Some features might be missing.")

if col == 'Ranking': # Ranking is critical for this section

st.error("Missing 'Ranking' column. Cannot proceed with ranking prediction.")

st.stop()

label\_enc\_ranking = LabelEncoder()

label\_enc\_concession = LabelEncoder()

label\_enc\_location = LabelEncoder()

if 'Ranking' in df.columns:

df['Ranking\_encoded'] = label\_enc\_ranking.fit\_transform(df['Ranking'])

else:

st.error("Missing 'Ranking' column for encoding. Cannot proceed.")

st.stop()

if 'Concession' in df.columns:

df['Concession\_encoded'] = label\_enc\_concession.fit\_transform(df['Concession'])

else:

st.warning("Missing 'Concession' column for encoding. Setting default to 0.")

df['Concession\_encoded'] = 0

if 'Location Name' in df.columns: # Use standardized name

df['Location\_encoded'] = label\_enc\_location.fit\_transform(df['Location Name'])

else:

st.warning("Missing 'Location Name' column for encoding. Setting default to 0.")

df['Location\_encoded'] = 0

# Features must use the standardized column names

features = ['Year', 'Concession\_encoded', 'Location\_encoded', 'Mathematics Index',

'Natural Sciences Index', 'Social And Citizenship Index', # Standardized name

'Critical Reading Index', 'English Index', 'Total Index']

# Filter features that are actually in the DataFrame

available\_features = [f for f in features if f in df.columns]

X = df[available\_features].dropna()

y = df.loc[X.index, 'Ranking\_encoded']

if X.empty or y.empty or len(y.unique()) < 2: # Need at least 2 classes for classification

st.warning("Not enough data or unique rankings to train the model after dropping NaNs and filtering features.")

st.stop()

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, stratify=y, random\_state=42)

xgb\_model = XGBClassifier(use\_label\_encoder=False, eval\_metric='mlogloss',

n\_estimators=100, max\_depth=5, random\_state=42)

xgb\_model.fit(X\_train, y\_train)

# --- Predict future years ---

latest = df[df['Year'] == df['Year'].max()].copy()

projected\_years = [2025, 2026, 2027]

projection\_data = []

for year in projected\_years:

future = latest.copy()

future['Year'] = year

if 'Concession' in future.columns:

future['Concession\_encoded'] = label\_enc\_concession.transform(future['Concession'])

else:

future['Concession\_encoded'] = 0 # Default if column missing

if 'Location Name' in future.columns: # Use standardized name

future['Location\_encoded'] = label\_enc\_location.transform(future['Location Name'])

else:

future['Location\_encoded'] = 0 # Default if column missing

X\_future = future[available\_features].dropna() # Use available\_features for consistency

future = future.loc[X\_future.index] # Realign 'future' with 'X\_future' indices after dropna

if not X\_future.empty:

predicted = xgb\_model.predict(X\_future)

future['Ranking\_encoded'] = predicted

future['Ranking'] = label\_enc\_ranking.inverse\_transform(predicted)

else:

future['Ranking\_encoded'] = np.nan

future['Ranking'] = "N/A"

future['Prediction'] = True

projection\_data.append(future)

projected\_df = pd.concat(projection\_data)

df['Prediction'] = False

combined\_df = pd.concat([df, projected\_df])

# --- Filter by type ---

type\_filter = st.sidebar.radio("🎓 Filter School Type", options=["All", "PUBLIC", "PRIVATE", "Concession"])

def filter\_data(df\_to\_filter): # Renamed parameter to avoid conflict

if type\_filter == "All":

return df\_to\_filter

elif type\_filter == "Concession":

if 'Concession' in df\_to\_filter.columns:

return df\_to\_filter[df\_to\_filter['Concession'].astype(str).str.upper() == 'YES']

return pd.DataFrame(columns=df\_to\_filter.columns) # Return empty df if column missing

else:

if 'Sector' in df\_to\_filter.columns:

return df\_to\_filter[df\_to\_filter['Sector'].astype(str).str.upper() == type\_filter.upper()]

return pd.DataFrame(columns=df\_to\_filter.columns) # Return empty df if column missing

filtered\_df = filter\_data(combined\_df)

# --- Timeline Plot ---

st.subheader("📊 Timeline of Rankings")

if 'Year' in filtered\_df.columns:

filtered\_df['Year'] = filtered\_df['Year'].astype(int)

real\_data = filtered\_df[filtered\_df['Prediction'] == False]

predict\_data = filtered\_df[filtered\_df['Prediction'] == True]

if 'Ranking' in real\_data.columns and not real\_data.empty:

timeline\_real = real\_data.groupby(['Year', 'Ranking']).size().unstack(fill\_value=0)

else:

timeline\_real = pd.DataFrame() # Ensure it's an empty DataFrame

if 'Ranking' in predict\_data.columns and not predict\_data.empty:

timeline\_predict = predict\_data.groupby(['Year', 'Ranking']).size().unstack(fill\_value=0)

else:

timeline\_predict = pd.DataFrame() # Ensure it's an empty DataFrame

fig = go.Figure()

if not timeline\_real.empty:

for col in timeline\_real.columns:

fig.add\_trace(go.Scatter(x=timeline\_real.index, y=timeline\_real[col],

mode='lines+markers', name=f"{col} (Real)"))

if not timeline\_predict.empty:

for col in timeline\_predict.columns:

fig.add\_trace(go.Scatter(x=timeline\_predict.index, y=timeline\_predict[col],

mode='lines+markers', name=f"{col} (Predicted)", line=dict(dash='dash')))

fig.update\_layout(title=f"Ranking Evolution ({type\_filter} Schools)",

xaxis\_title='Year', yaxis\_title='Number of Schools', height=500)

st.plotly\_chart(fig, use\_container\_width=True)

# --- 2027 Predicted Ranking ---

st.subheader("🔮 Predicted Rankings for 2027")

pred\_2027 = projected\_df[projected\_df['Year'] == 2027]

pred\_2027\_filtered = filter\_data(pred\_2027)

if not pred\_2027\_filtered.empty and 'Ranking' in pred\_2027\_filtered.columns:

count\_df = pred\_2027\_filtered['Ranking'].value\_counts().reset\_index()

count\_df.columns = ['Ranking', 'Count']

fig\_bar = px.bar(count\_df, x='Ranking', y='Count', color='Ranking', text='Count',

title=f"Predicted Rankings for 2027 ({type\_filter} Schools)")

fig\_bar.update\_traces(textposition='outside')

fig\_bar.update\_layout(height=400)

st.plotly\_chart(fig\_bar, use\_container\_width=True)

else:

st.info("No predicted ranking data for 2027 based on current filters.")

# --- Details Table ---

st.markdown("### 🏢 School List")

# Ensure columns exist before displaying

# Use "Name Of Establishment" (standardized)

display\_cols = ["Name Of Establishment", 'City', 'Sector', 'Concession', 'Ranking']

display\_cols\_present = [col for col in display\_cols if col in pred\_2027\_filtered.columns]

if not pred\_2027\_filtered.empty and 'Ranking' in pred\_2027\_filtered.columns: # Check if not empty and 'Ranking' exists

if "Name Of Establishment" not in pred\_2027\_filtered.columns:

st.warning("The column 'Name Of Establishment' is missing in the filtered data for display.")

st.dataframe(pred\_2027\_filtered[display\_cols\_present].sort\_values('Ranking'))

else:

st.info("No schools to display in the list.")

# --- Highlight C and D ---

st.subheader("🚨 Schools Predicted in C or D")

if 'Ranking' in pred\_2027\_filtered.columns:

c\_or\_d = pred\_2027\_filtered[

pred\_2027\_filtered['Ranking'].isin(['C', 'D'])].copy() # Add .copy() to avoid SettingWithCopyWarning

else:

c\_or\_d = pd.DataFrame(columns=pred\_2027\_filtered.columns) # Create an empty DataFrame with same columns

if not c\_or\_d.empty and 'Ranking' in c\_or\_d.columns:

c\_or\_d\_count = c\_or\_d['Ranking'].value\_counts().reset\_index()

c\_or\_d\_count.columns = ['Ranking', 'Count']

fig\_cd = px.bar(c\_or\_d\_count, x='Ranking', y='Count', color='Ranking', text='Count',

title="Schools Predicted in C or D")

fig\_cd.update\_traces(textposition='outside')

st.plotly\_chart(fig\_cd, use\_container\_width=True)

# Ensure display columns are present in c\_or\_d

display\_cols\_cd = [col for col in display\_cols if col in c\_or\_d.columns]

st.dataframe(c\_or\_d[display\_cols\_cd])

else:

st.info("No schools predicted in C or D for 2027 based on current filters.")

# --- Location Breakdown of Public C/D ---

st.subheader("📍 Public Schools in C or D (by Location)")

# Check for all necessary columns before filtering

if all(col in pred\_2027.columns for col in ['Concession', 'Ranking', 'Location Name', 'Sector']):

public\_cd = pred\_2027[

(pred\_2027['Sector'].astype(str).str.upper() == 'PUBLIC') & # Changed 'Concession' to 'Sector' for public

(pred\_2027['Ranking'].isin(['C', 'D']))].copy() # Add .copy()

if not public\_cd.empty:

loc\_count = public\_cd['Location Name'].value\_counts().reset\_index() # Use standardized 'Location Name'

loc\_count.columns = ['Location Name', 'Count'] # Use standardized 'Location Name'

fig\_loc = px.bar(loc\_count, x='Count', y='Location Name', orientation='h',

# Use standardized 'Location Name'

title="📍 Public Schools in C or D by Location",

color='Count', color\_continuous\_scale='YlOrRd')

fig\_loc.update\_layout(height=500)

st.plotly\_chart(fig\_loc, use\_container\_width=True)

else:

st.info("No public schools predicted in C or D for 2027.")

else:

st.info("Required columns ('Sector', 'Ranking', or 'Location Name') are missing for this analysis.")

# --- Top Feature Importance for Public C/D ---

st.subheader("🔍 Top Predictors for C/D Ranked Public Schools")

# Ensure public\_cd is not empty and has enough unique ranking values for a meaningful model

if 'Ranking\_encoded' in public\_cd.columns and not public\_cd.empty: # Removed len(unique()) > 1 check here, moved below

# Filter for available\_features here too for X\_public\_cd

X\_public\_cd = public\_cd[[f for f in features if f in public\_cd.columns]].dropna() # Use a list comprehension

# Check if X\_public\_cd is not empty after dropna

if not X\_public\_cd.empty:

y\_public\_cd\_original = public\_cd.loc[X\_public\_cd.index, 'Ranking\_encoded'] # Keep original encoded values

# Check for at least two unique classes AFTER filtering for C/D and dropping NaNs

if len(y\_public\_cd\_original.unique()) < 2:

st.info("Not enough variety in C/D rankings to calculate feature importance for public C/D schools.")

else:

# FIX: Re-encode y\_public\_cd to be binary (0 or 1)

# Map the original encoded values (e.g., 3 and 4) to 0 and 1

# This ensures the target is binary for temp\_xgb\_model

# First, find the mapping of 'C' and 'D' to their original encoded values

# You might have A+=0, A=1, B=2, C=3, D=4

# Or A+=4, A=3, B=2, C=1, D=0

# A robust way is to use a new LabelEncoder specifically for 'C' and 'D'

binary\_label\_encoder = LabelEncoder()

# Use the \*original\* 'Ranking' column for fitting the new encoder

# It's safer to work with actual labels here, then map.

# Get the actual string labels for C and D from the original label\_encoder

c\_label\_val = \

[k for k, v in zip(label\_enc\_ranking.classes\_, label\_enc\_ranking.transform(label\_enc\_ranking.classes\_))

if k == 'C'][0]

d\_label\_val = \

[k for k, v in zip(label\_enc\_ranking.classes\_, label\_enc\_ranking.transform(label\_enc\_ranking.classes\_))

if k == 'D'][0]

# Create a mapping from original encoded values (3, 4) to new binary (0, 1)

# Assign the lower encoded value to 0 and the higher to 1.

# Or, if you want 'C' to always be 0 and 'D' to always be 1, regardless of their original numeric order:

# Option 1: Consistent mapping for 'C' and 'D'

mapping\_dict = {

label\_enc\_ranking.transform(['C'])[0]: 0, # Map original encoded 'C' to 0

label\_enc\_ranking.transform(['D'])[0]: 1 # Map original encoded 'D' to 1

}

y\_public\_cd = y\_public\_cd\_original.map(mapping\_dict)

# Now, fit the temp\_xgb\_model

temp\_xgb\_model = XGBClassifier(use\_label\_encoder=False, eval\_metric='logloss',

# Use 'logloss' for binary classification

n\_estimators=100, random\_state=42)

temp\_xgb\_model.fit(X\_public\_cd, y\_public\_cd)

booster = temp\_xgb\_model.get\_booster()

importance\_dict = booster.get\_score(importance\_type='gain')

if importance\_dict: # Check if importance\_dict is not empty

importance\_df = pd.DataFrame({

'Feature': list(importance\_dict.keys()),

'Importance': list(importance\_dict.values())

}).sort\_values('Importance', ascending=True)

# Normalize importance scores if needed (optional, for better visualization)

importance\_df['Importance'] /= importance\_df['Importance'].sum()

# Map original feature names to user-friendly names

feature\_mapping = {

'Concession\_encoded': 'Concession',

'Location\_encoded': 'Location',

'Mathematics Index': 'Mathematics Index',

'Natural Sciences Index': 'Natural Sciences Index',

'Social And Citizenship Index': 'Social And Citizenship Index',

'Critical Reading Index': 'Critical Reading Index',

'English Index': 'English Index',

'Total Index': 'Total Index',

'Year': 'Year'

}

importance\_df['Feature\_Display'] = importance\_df['Feature'].replace(feature\_mapping)

fig\_feat = px.bar(importance\_df, x='Importance', y='Feature\_Display', orientation='h',

title="🔍 Gain-Based Feature Importance (C/D Public Schools)",

color='Importance', color\_continuous\_scale='Blues')

fig\_feat.update\_layout(height=500)

st.plotly\_chart(fig\_feat, use\_container\_width=True)

else:

st.info("No feature importance data to display for public C/D schools (all importances are zero).")

else:

st.info("Not enough data to calculate feature importance for public C/D schools after filtering.")

else:

st.info("No data available or insufficient ranking variety to show feature importance for public C/D schools.")

Prescriptive

import streamlit as st

import pandas as pd

import numpy as np

import copy

from sklearn.preprocessing import MinMaxScaler

from pulp import LpProblem, LpVariable, lpSum, LpMaximize, LpInteger

import plotly.express as px

# ------------------ Page Setup ------------------

st.set\_page\_config(page\_title="Strategic Budget Optimizer", layout="wide")

st.title("📊 Strategic Budget Optimizer")

# ------------------ Section Selection ------------------

section = st.sidebar.selectbox(

"Select Section",

["1. School Investment Prioritization", "2. Budget Allocation", "3. Funding Simulation"]

)

# ------------------ Base Data ------------------

default\_data = {

"Infrastructure": {

"Primary Classrooms": {"cost": 4999.0, "weight": 0.12},

"TIM Classroom": {"cost": 11253.0, "weight": 0.08},

"Science Lab (Primary)": {"cost": 9451.0, "weight": 0.07},

"Library": {"cost": 11976.0, "weight": 0.11},

"Bilingualism Room": {"cost": 14790.0, "weight": 0.04},

"Reading Room": {"cost": 1639.0, "weight": 0.03},

"Workstations": {"cost": 600.0, "weight": 0.175},

"Science Lab Workstations": {"cost": 700.0, "weight": 0.07},

"Shared Spaces": {"cost": 1250.0, "weight": 0.075},

"Storage & Lockers": {"cost": 360.0, "weight": 0.03},

"Kitchen & Dining": {"cost": 3.0, "weight": 0.03}

},

"Training": {

"Digital Pedagogy": {"cost": 160.0, "weight": 0.25},

"STEM/STEAM Instruction": {"cost": 250.0, "weight": 0.30},

"School Leadership": {"cost": 200.0, "weight": 0.20},

"Inclusion & SEL": {"cost": 150.0, "weight": 0.25},

},

"Technology": {

"Educational Software & Licenses": {"cost": 100.0, "weight": 0.30},

"Network Infrastructure": {"cost": 600.0, "weight": 0.25},

"Audio/Visual Tools": {"cost": 250.0, "weight": 0.20},

"Cybersecurity": {"cost": 375.0, "weight": 0.15},

"IT Maintenance & Support": {"cost": 250.0, "weight": 0.10},

},

"Salaries": {

"Base Teacher": {"cost": 15000.0, "weight": 0.1887},

"Specialized Teacher": {"cost": 24000.0, "weight": 0.2830},

"Administrator": {"cost": 10000.0, "weight": 0.1509},

"Manager": {"cost": 28000.0, "weight": 0.3774},

}

}

category\_data = copy.deepcopy(default\_data)

# ------------------ Section 1: TOPSIS School Prioritization ------------------

if section == "1. School Investment Prioritization":

st.header("🏢 School Investment Prioritization")

# --- Load Sample Data ---

st.markdown("### 🎓 Sample Prioritization Result (TOPSIS)")

sample\_df = None

try:

sample\_df = pd.read\_csv(r"C:\Study document\6611 Dataset\Final\sampled\_schools\_model.csv")

except FileNotFoundError:

st.warning("⚠️ Sample file not found. Please upload your own CSV file below to see the prioritization results.")

if sample\_df is not None:

sample\_df.columns = sample\_df.columns.str.strip().str.replace(" ", "\_")

indicators = [

"Mathematics\_Index", "Natural\_Sciences\_Index", "Social\_and\_Citizenship\_Index",

"Critical\_Reading\_Index", "English\_Index",

"Aprobacion\_(Passrate)", "Reprobacion\_(Fail\_rate)", "Descercion\_(Dropoutrate)"

]

for col in ["Aprobacion\_(Passrate)", "Reprobacion\_(Fail\_rate)", "Descercion\_(Dropoutrate)"]:

if sample\_df[col].dtype == object:

sample\_df[col] = sample\_df[col].str.replace('%', '').astype(float) / 100

weights = [0.15, 0.15, 0.15, 0.15, 0.15, 0.10, 0.10, 0.05]

benefit\_criteria = {

"Mathematics\_Index": True, "Natural\_Sciences\_Index": True, "Social\_and\_Citizenship\_Index": True,

"Critical\_Reading\_Index": True, "English\_Index": True,

"Aprobacion\_(Passrate)": True, "Reprobacion\_(Fail\_rate)": False, "Descercion\_(Dropoutrate)": False

}

# Normalize and Apply TOPSIS

scaler = MinMaxScaler()

normalized = pd.DataFrame(scaler.fit\_transform(sample\_df[indicators]), columns=indicators)

for col in indicators:

if not benefit\_criteria[col]:

normalized[col] = 1 - normalized[col]

weighted = normalized \* weights

ideal\_best = weighted.max()

ideal\_worst = weighted.min()

distance\_best = np.sqrt(((weighted - ideal\_best) \*\* 2).sum(axis=1))

distance\_worst = np.sqrt(((weighted - ideal\_worst) \*\* 2).sum(axis=1))

sample\_df["TOPSIS\_Score"] = distance\_worst / (distance\_best + distance\_worst)

sample\_df["Rank"] = sample\_df["TOPSIS\_Score"].rank(ascending=False)

top\_sample = sample\_df.sort\_values("TOPSIS\_Score", ascending=False)[[

"Name\_of\_Establishment", "TOPSIS\_Score", "Rank"

]].reset\_index(drop=True)

st.dataframe(top\_sample.head(10), use\_container\_width=True)

# Bar chart instead of scatter to avoid label overlap

fig = px.bar(

top\_sample.head(10).sort\_values("TOPSIS\_Score"),

x="TOPSIS\_Score", y="Name\_of\_Establishment",

orientation="h", color="TOPSIS\_Score",

color\_continuous\_scale="Blues",

title="🔍 Sample: TOPSIS Score of Top 10 Schools"

)

fig.update\_layout(xaxis\_title="TOPSIS Score", yaxis\_title="School Name")

st.plotly\_chart(fig, use\_container\_width=True)

# --- File Upload Option ---

st.markdown("---")

st.markdown("### 📂 Upload Your Own Dataset to Run TOPSIS")

uploaded\_file = st.file\_uploader("Upload a CSV file", type=["csv"])

if uploaded\_file:

df = pd.read\_csv(uploaded\_file)

df.columns = df.columns.str.strip().str.replace(" ", "\_")

for col in ["Aprobacion\_(Passrate)", "Reprobacion\_(Fail\_rate)", "Descercion\_(Dropoutrate)"]:

if df[col].dtype == object:

df[col] = df[col].str.replace('%', '').astype(float) / 100

normalized = pd.DataFrame(scaler.fit\_transform(df[indicators]), columns=indicators)

for col in indicators:

if not benefit\_criteria[col]:

normalized[col] = 1 - normalized[col]

weighted = normalized \* weights

ideal\_best = weighted.max()

ideal\_worst = weighted.min()

distance\_best = np.sqrt(((weighted - ideal\_best) \*\* 2).sum(axis=1))

distance\_worst = np.sqrt(((weighted - ideal\_worst) \*\* 2).sum(axis=1))

df["TOPSIS\_Score"] = distance\_worst / (distance\_best + distance\_worst)

df["Rank"] = df["TOPSIS\_Score"].rank(ascending=False)

top\_user = df.sort\_values("TOPSIS\_Score", ascending=False)[[

"Name\_of\_Establishment", "TOPSIS\_Score", "Rank"

]].reset\_index(drop=True)

st.success("✅ TOPSIS Completed for Uploaded File!")

st.dataframe(top\_user.head(10), use\_container\_width=True)

fig2 = px.bar(

top\_user.head(10).sort\_values("TOPSIS\_Score"),

x="TOPSIS\_Score", y="Name\_of\_Establishment",

orientation="h", color="TOPSIS\_Score",

color\_continuous\_scale="Viridis",

title="🏫 Uploaded: TOPSIS Score of Top 10 Schools"

)

fig2.update\_layout(xaxis\_title="TOPSIS Score", yaxis\_title="School Name")

st.plotly\_chart(fig2, use\_container\_width=True)

# ------------------ Inputs for Budget Sections ------------------

if section in ["2. Budget Allocation", "3. Funding Simulation"]:

total\_budget = st.sidebar.number\_input("💰 Total 10-Year Budget (USD)", value=1\_219\_399.66, step=10000.0,

format="%.2f")

num\_students = st.sidebar.number\_input("👩‍🏫 Number of Students in Concession Program", min\_value=0, step=1)

st.sidebar.subheader("📌 Main Category Weights")

raw\_weights = {

cat: st.sidebar.slider(f"{cat} Weight (%)", 0, 100, int(100 \* sum(sub["weight"] for sub in items.values())),

key=f"main\_{cat}")

for cat, items in category\_data.items()

}

total\_main\_weight = sum(raw\_weights.values())

normalized\_main\_weights = {k: v / total\_main\_weight for k, v in raw\_weights.items()}

allocated\_budget = {k: total\_budget \* normalized\_main\_weights[k] for k in category\_data}

st.sidebar.subheader("⚙️ Edit Sub-Items (Weight & Cost)")

for cat, items in category\_data.items():

with st.sidebar.expander(f"🔧 {cat}"):

for item\_name in items:

item = items[item\_name]

item["cost"] = st.number\_input(f"{item\_name} - Cost", min\_value=0.0, value=float(item["cost"]),

step=100.0, format="%.2f")

weight\_percent = st.slider(f"{item\_name} - Weight (%)", 0.0, 100.0,

value=round(100 \* item["weight"], 2), step=1.0)

item["weight"] = weight\_percent / 100.0

# ------------------ Optimization Function ------------------

def run\_optimization(data, budget, num\_students):

model = LpProblem("Optimize", LpMaximize)

safe\_keys = {k: k.replace(" ", "\_").replace("(", "").replace(")", "") for k in data}

x = {safe\_keys[k]: LpVariable(safe\_keys[k], lowBound=0, cat=LpInteger) for k in data}

model += lpSum(x[safe\_keys[k]] \* data[k]["weight"] for k in data)

model += lpSum(x[safe\_keys[k]] \* data[k]["cost"] for k in data) <= budget

bt = st = admin = mgr = None

if "Primary Classrooms" in data:

pc = safe\_keys["Primary Classrooms"]

min\_classrooms = int(np.ceil(num\_students / 40))

model += x[pc] >= min\_classrooms

model += x[pc] <= min\_classrooms + 10

if "TIM Classroom" in data and "Primary Classrooms" in data:

model += x[safe\_keys["TIM Classroom"]] >= 0.5 \* x[safe\_keys["Primary Classrooms"]]

if all(k in data for k in ["Base Teacher", "Specialized Teacher", "Primary Classrooms"]):

bt = safe\_keys["Base Teacher"]

st = safe\_keys["Specialized Teacher"]

pc = safe\_keys["Primary Classrooms"]

model += x[bt] >= 10

model += x[bt] <= x[pc]

model += x[st] >= 10

model += x[st] < x[pc]

model += x[bt] + x[st] >= x[pc]

model += x[bt] + x[st] <= x[pc] + 10

model += x[st] >= 0.5 \* x[bt]

if "Administrator" in data:

admin = safe\_keys["Administrator"]

model += x[admin] >= 5

model += x[admin] <= 10

if "Manager" in data:

mgr = safe\_keys["Manager"]

model += x[mgr] >= 3

model += x[mgr] <= 10

if all(v is not None for v in [bt, st, admin, mgr]):

model += x[mgr] + x[admin] <= x[bt] + x[st] - 1

for k in ["Digital Pedagogy", "STEM/STEAM Instruction", "School Leadership", "Inclusion & SEL"]:

if k in data:

key = safe\_keys[k]

model += x[key] >= 40

model += x[key] <= 199

if "Kitchen & Dining" in data:

model += x[safe\_keys["Kitchen & Dining"]] <= num\_students / 2

if "Storage & Lockers" in data:

model += x[safe\_keys["Storage & Lockers"]] >= num\_students / 3

model += x[safe\_keys["Storage & Lockers"]] <= num\_students / 2

if "Library" in data and "Primary Classrooms" in data:

model += x[safe\_keys["Library"]] >= 5

model += x[safe\_keys["Library"]] <= x[safe\_keys["Primary Classrooms"]]

for room in [

"Science Lab (Primary)", "Library", "Bilingualism Room", "Reading Room",

"Workstations", "Science Lab Workstations", "Shared Spaces"

]:

if room in data:

rk = safe\_keys[room]

model += x[rk] >= 5

model += x[rk] <= 10

for k, (lb, ub) in {

"Educational Software & Licenses": (0, 400),

"Network Infrastructure": (20, 400),

"Audio/Visual Tools": (20, 400),

"Cybersecurity": (20, 400),

"IT Maintenance & Support": (20, 400)

}.items():

if k in data:

model += x[safe\_keys[k]] >= lb

model += x[safe\_keys[k]] <= ub

model.solve()

result = {k: max(0, int(round(x[safe\_keys[k]].varValue or 0))) for k in data}

total\_used\_cost = sum(result[k] \* data[k]["cost"] for k in data)

return result, total\_used\_cost

# ------------------ Allocation Renderer ------------------

def render\_allocation(category\_data, allocated\_budget, num\_students, simulation=False, num\_sim=100):

for category, items in category\_data.items():

st.subheader(f"📦 {category}")

budget = allocated\_budget[category]

costs = []

for i in range(num\_sim if simulation else 1):

perturbed = {

k: {

"cost": v["cost"] \* np.random.uniform(0.9, 1.1) if simulation else v["cost"],

"weight": v["weight"]

} for k, v in items.items()

}

result, used\_cost = run\_optimization(perturbed, budget, num\_students)

costs.append(used\_cost)

if not simulation:

df = pd.DataFrame(result.items(), columns=["Item", "Units"])

df["Cost per Unit"] = df["Item"].map({k: v["cost"] for k, v in items.items()})

df["Weight"] = df["Item"].map({k: v["weight"] for k, v in items.items()})

df["Total Cost"] = df["Units"] \* df["Cost per Unit"]

st.markdown(f"\*\*Allocated Budget:\*\* ${budget:,.2f}")

st.markdown(f"\*\*Used Cost:\*\* ${used\_cost:,.2f}")

st.dataframe(df, use\_container\_width=True)

fig = px.bar(df, x="Item", y="Units", text="Units", title=f"{category} - Allocated Units")

st.plotly\_chart(fig, use\_container\_width=True)

if simulation:

st.metric("Mean Cost", f"${np.mean(costs):,.2f}")

st.metric("Std Dev", f"${np.std(costs):,.2f}")

fig = px.histogram(costs, nbins=30, title=f"{category} - Cost Distribution")

st.plotly\_chart(fig, use\_container\_width=True)

# ------------------ Render Budget Sections ------------------

if section == "2. Budget Allocation":

st.header("📊 Budget Allocation")

render\_allocation(category\_data, allocated\_budget, num\_students)

elif section == "3. Funding Simulation":

st.header("🌾 Monte Carlo Funding Simulation")

num\_simulations = st.number\_input(

"Number of Simulations",

min\_value=100,

max\_value=5000,

value=100, # ✅ Set default to 100

step=100

)

render\_allocation(category\_data, allocated\_budget, num\_students, simulation=True, num\_sim=num\_simulations)