1. Code for data cleaning, testing, training and model selection

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LinearRegression
from sklearn.neighbors import KNeighborsRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor, AdaBoostRegressor, GradientBoostingRegressor
from xgboost import XGBRegressor
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.metrics import f1_score
df = pd.read_csv('/content/IndicatorS.csv')
df = df.dropna()
X = df[['CountryName', 'CountryCode', 'IndicatorName', 'IndicatorCode', 'Year']]
y = df['Value']
categorical_features = ['CountryName', 'CountryCode', 'IndicatorName', 'IndicatorCode']
numerical_features = ['Year']
preprocessor = ColumnTransformer(transformers=[
    ('cat', OneHotEncoder(handle_unknown='ignore'), categorical_features),
    ('num', StandardScaler(), numerical_features)
])
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
model_1 = Pipeline(steps=[('pre', preprocessor), ('reg', LinearRegression())])
model_2 = Pipeline(steps=[('pre', preprocessor), ('reg', KNeighborsRegressor())])
model_3 = Pipeline(steps=[('pre', preprocessor), ('reg', DecisionTreeRegressor(random_state=42))])
model_4 = Pipeline(steps=[('pre', preprocessor), ('reg', RandomForestRegressor(n_estimators=100,
random_state=42))])
model_5 = Pipeline(steps=[('pre', preprocessor), ('reg', XGBRegressor(n_estimators=100, random_state=42))])
model_6 = Pipeline(steps=[('pre', preprocessor), ('reg', AdaBoostRegressor(n_estimators=100,
random_state=42))])
model_7 = Pipeline(steps=[('pre', preprocessor), ('reg', GradientBoostingRegressor(n_estimators=100,
random_state=42))])
model_1.fit(X_train, y_train)
model_2.fit(X_train, y_train)
model_3.fit(X_train, y_train)
model_4.fit(X_train, y_train)
model_5.fit(X_train, y_train)
model_6.fit(X_train, y_train)
model_7.fit(X_train, y_train)
pred_1 = model_1.predict(X_test)
pred_2 = model_2.predict(X_test)
pred_3 = model_3.predict(X_test)
```

```
                                pred_4 = model_4.predict(X_test)
pred_5 = model_5.predict(X_test)
pred_6 = model_6.predict(X_test)
pred_7 = model_7.predict(X_test)
def evaluate_model(name, y_true, y_pred):
    mse = mean_squared_error(y_true, y_pred)
    rmse = np.sqrt(mse)
    r2 = r2_score(y_true, y_pred)
    print(f"\n{name}")
    print(f"R² Score: {r2:.4f}")
    print(f"RMSE    : {rmse:.4f}")
evaluate_model("Model 1: Linear Regression", y_test, pred_1)
evaluate_model("Model 2: KNN", y_test, pred_2)
evaluate_model("Model 3: Decision Tree", y_test, pred_3)
evaluate_model("Model 4: Random Forest", y_test, pred_4)
evaluate_model("Model 5: XGBoost", y_test, pred_5)
evaluate_model("Model 6: AdaBoost", y_test, pred_6)
evaluate_model("Model 7: Gradient Boost", y_test, pred_7)
# Define models dictionary BEFORE the loop
models = {
    "Linear Regression": LinearRegression(),
    "KNN": KNeighborsRegressor(),
    "Decision Tree": DecisionTreeRegressor(random_state=42),
    "Random Forest": RandomForestRegressor(n_estimators=100, random_state=42),
    "XGBoost": XGBRegressor(n_estimators=100, random_state=42, verbosity=0),
    "AdaBoost": AdaBoostRegressor(n_estimators=100, random_state=42),
    "Gradient Boost": GradientBoostingRegressor(n_estimators=100, random_state=42)
}
bins = [0, 100000, 500000, float('inf')]
labels = ['Low', 'Medium', 'High']

regression_results = []
classification_reports = {}
confusion_matrices = {}

for name, regressor in models.items():
    print(f"\n======= {name} =======")

    # Build pipeline
    model = Pipeline([('pre', preprocessor), ('reg', regressor)])
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)

    # Regression evaluation
    r2 = r2_score(y_test, y_pred)
    rmse = np.sqrt(mean_squared_error(y_test, y_pred))
    print(f"R² Score: {r2:.4f}")
    print(f"RMSE    : {rmse:.2f}")
    regression_results.append((name, r2, rmse))

    # Simulated classification
    y_true_binned = pd.cut(y_test, bins=bins, labels=labels)
```

```
y_pred_binned = pd.cut(y_pred, bins=bins, labels=labels)

    # 🔒 Filter out any NaNs (due to values outside bin ranges)
    mask = (~y_true_binned.isna()) & (~y_pred_binned.isna())
    y_true_binned_clean = y_true_binned[mask]
    y_pred_binned_clean = y_pred_binned[mask]

    # Classification report
    report = classification_report(
        y_true_binned_clean,
        y_pred_binned_clean,
        labels=labels,
        output_dict=False,
        zero_division=0
    )

    print("\nClassification Report:")
    print(report)

    # Confusion matrix
    cm = confusion_matrix(y_true_binned_clean, y_pred_binned_clean, labels=labels)
    print("\nConfusion Matrix:")
    print(cm)

    classification_reports[name] = report
    confusion_matrices[name] = cm

    # Plot confusion matrix
    plt.figure(figsize=(5, 4))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=labels, yticklabels=labels)
    plt.title(f"Confusion Matrix - {name}")
    plt.xlabel("Predicted")
    plt.ylabel("Actual")
    plt.tight_layout()
    plt.show()
# Filter original df to find CO2-related indicators
co2_indicators = df[df['IndicatorName'].str.contains('CO2', case=False)][['IndicatorName',
'IndicatorCode']].drop_duplicates()
print(co2_indicators)
# Use a CO2-specific indicator (modify if you prefer another)
selected_indicator_name = "CO2 emissions (metric tons per capita)"
selected_indicator_code = "EN.ATM.CO2E.PC"
df = df[df['IndicatorName'].str.contains('CO2', case=False)]
def predict_co2_emission(country_name, year):
    # Get the country code
    try:
        country_code = df[df['CountryName'] == country_name]['CountryCode'].iloc[0]
    except IndexError:
        return f"❌ Country '{country_name}' not found in the dataset."

    # Create input row
    input_data = pd.DataFrame([{
```

```
                                    'CountryName': country_name,
        'CountryCode': country_code,
        'IndicatorName': selected_indicator_name,
        'IndicatorCode': selected_indicator_code,
        'Year': year
    }])

    # Predict using trained XGBoost model
    prediction = model_5.predict(input_data)[0]

    returnf" Predicted CO₂ emission for {country_name} in {year}: {prediction:.4f} metric tons per capita"
```

2. Code for creating pickle file and ngrok installation with HTML code in it with app.run().

```
from flask import Flask, request, render_template_string
from flask_ngrok import run_with_ngrok
import pickle
import numpy as np

# Load your model and encoder
with open("co2_rf_model.pkl", "rb") as f:
    model = pickle.load(f)

with open("label_encoder.pkl", "rb") as f:
    le = pickle.load(f)

# HTML template
html = '''
<!DOCTYPE html>
<html>
<head><title>CO₂ Predictor</title></head>
<body>
    <h2>Predict CO₂ Emissions (kt)</h2>
    <form method="POST">
        Country: <input name="country"><br><br>
        Year: <input name="year" type="number"><br><br>
        <input type="submit" value="Predict">
    </form>
    {% if prediction %}
    <p><strong>Prediction:</strong> {{ prediction | round(2) }} kt</p>
    {% elif error %}
    <p style="color:red">{{ error }}</p>
    {% endif %}
</body>
</html>
'''

# Setup Flask
app = Flask(_name_)
run_with_ngrok(app)

@app.route("/", methods=["GET", "POST"])
```

```
def index():
    prediction = None
    error = None
    if request.method == "POST":
        country = request.form["country"]
        year = request.form["year"]

        try:
            year = int(year)
            if country not in le.classes_:
                error = f"'{country}' not found in training data."
            else:
                encoded_country = le.transform([country])[0]
                x = np.array([[encoded_country, year]])
                prediction = model.predict(x)[0]
        except Exception as e:
            error = str(e)

    return render_template_string(html, prediction=prediction, error=error)

app.run()
```