ETL Project — Student Dataset (Jupyter-style notebook)

Goal: Clean, transform, and load the `student` dataset into a PostgreSQL database. This notebook is written as sequential Jupyter cells. Each cell has a short **Markdown explanation** (simple English) and then the **Python code** to run. Use this notebook to post on social media or to run locally.

About the columns

Below are simple explanations for the columns you listed. Use these descriptions in captions or the project README when you share.

- **school**: which school the student attends (short code, e.g., `GP` or `MS`).
- **sex**: student gender (`F` = female, `M` = male).
- **age**: student age (years, integer).
- **address**: home address type (`U` = urban, `R` = rural).
- **famsize**: family size (`LE3` = less or equal 3, `GT3` = greater than 3).
- **Pstatus**: parent cohabitation status (`T` = living together, `A` = apart).
- **Medu**: mother's education (0 to 4 scale, where 4 is highest).
- **Fedu**: father's education (0 to 4 scale).
- **Mjob**: mother's job (categories: `teacher`, `health`, `services`, `at_home`, `other`).
- **Fjob**: father's job (same possible categories as `Mjob`).
- **reason**: reason to choose this school (e.g., `home`, `reputation`, `course`, `other`).
- **guardian**: student's guardian (`mother`, `father`, `other`).
- **traveltime**: home to school travel time (1 to 4, categorical numeric).
- **studytime**: weekly study time (1 to 4, categorical numeric).
- **failures**: number of past class failures (integer).
- **schoolsup**: extra educational support (`yes` or `no`).
- **famsup**: family educational support (`yes` or `no`).
- **paid**: extra paid classes (`yes` or `no`).
- **activities**: participates in extracurricular activities (`yes` or `no`).
- **nursery**: attended nursery school (`yes` or `no`).
- **higher**: wants to take higher education (`yes` or `no`).
- **internet**: internet access at home (`yes` or `no`).
- **romantic**: in a romantic relationship (`yes` or `no`).
- **famrel**: quality of family relationships (1 - very bad to 5 - excellent).
- **freetime**: free time after school (1 to 5).
- **goout**: how often goes out with friends (1 to 5).
- **Dalc**: workday alcohol consumption (1 to 5).
- **Walc**: weekend alcohol consumption (1 to 5).
- **health**: current health status (1 to 5).
- **absences**: number of school absences.

- **G1**: first period grade (numeric, 0-20).
- **G2**: second period grade (numeric, 0-20).
- **G3**: final grade (numeric, 0-20) — often considered the target variable.

Notebook cells

Cell 1 — Setup: install packages (if needed) and import libraries

Explanation (simple English): - We make sure necessary Python packages are available. - We import the packages we will use for cleaning and loading data: `pandas` for data work, `numpy` for numbers, `sqlalchemy` and `psycopg2` to talk to PostgreSQL, and `python-dotenv` to load safe credentials from a `.env` file.

```
# If you run this notebook on a fresh environment, uncomment the next line to
install packages.
# In Jupyter use %pip; that keeps the kernel environment consistent.

# %pip install pandas numpy sqlalchemy psycopg2-binary python-dotenv

# Import standard libraries
import os
import pandas as pd
import numpy as np
from sqlalchemy import create_engine, types
from dotenv import load_dotenv

# Load environment variables from a .env file (we configure this later)
load_dotenv()
```

Cell 2 — Load the dataset

Explanation: - Read the CSV file named `student.csv` into a pandas DataFrame named `df`. - Show the number of rows and columns and display the first 5 rows to inspect.

```
# Replace the path if your CSV is in a different folder
csv_path = "student.csv"

# Read the CSV file
df = pd.read_csv(csv_path)

# Print shape (rows, columns) and show first 5 rows
```

```
print("Shape:", df.shape)
df.head()
```

Cell 3 — Quick data inspection

Explanation: - `info()` shows column data types and non-null counts. - `describe(include='all')` gives summary statistics for both numeric and non-numeric columns. - `nunique()` helps find how many unique values each column has. - `isnull().sum()` shows how many missing values per column.

```
# Basic info
print("--- INFO ---")
df.info()

print("\n--- DESCRIBE (all columns) ---")
print(df.describe(include='all'))

print("\n--- Unique counts ---")
print(df.nunique())

print("\n--- Missing values ---")
print(df.isnull().sum())
```

Cell 4 — Normalize column names

Explanation: - Make column names lowercase, remove leading/trailing spaces, and replace spaces with underscores. - This makes column names consistent and easier to reference in code.

```
# Normalize column names
original_columns = df.columns.tolist()
df.columns = (df.columns
              .str.strip()
              .str.lower()
              .str.replace(' ', '_'))

print("Original columns:\n", original_columns)
print("\nNormalized columns:\n", df.columns.tolist())
```

Cell 5 — Convert numeric columns to proper numeric types

Explanation: - Some columns may be read as strings even though they hold numbers (because of bad values or commas). - We convert the most likely numeric columns to numeric types using `pd.to_numeric`. - `errors='coerce'` turns invalid values into `NaN`, which we will handle later.

```
# Candidate numeric columns
num_cols = ['age','medu','fedu','traveltime','studytime','failures',
            'famrel','freetime','goout','dalc','walc','health',
            'absences','g1','g2','g3']

for col in num_cols:
    if col in df.columns:
        df[col] = pd.to_numeric(df[col], errors='coerce')

# Confirm changes
df[num_cols].dtypes
```

Cell 6 — Remove exact duplicate rows (if any)

Explanation: - Drop rows that are exact duplicates across all columns. - Report how many duplicates were found and removed.

```
before = df.shape[0]
df = df.drop_duplicates()
after = df.shape[0]
print(f"Dropped {before - after} exact duplicate rows.")
```

Cell 7 — Handle missing values (strategy)

Explanation (simple): - For numeric columns, fill missing values with the *median* (robust to outliers). - For categorical columns, fill missing values with the *mode* (most common value). - You can change strategy later (e.g., drop rows, use KNN imputation, domain-specific values).

```
# Split columns into numeric and categorical sets (only columns that exist)
existing_num_cols = [c for c in num_cols if c in df.columns]
existing_cat_cols = [c for c in df.columns if c not in existing_num_cols]

# Fill numeric columns with median
for c in existing_num_cols:
    median_val = df[c].median()
```

```
        df[c] = df[c].fillna(median_val)

# Fill categorical columns with mode (if mode exists)
for c in existing_cat_cols:
    if df[c].isnull().any():
        try:
            mode_val = df[c].mode().iloc[0]
            df[c] = df[c].fillna(mode_val)
        except IndexError:
            # Column might be empty — fill with placeholder
            df[c] = df[c].fillna('Unknown')

# Quick check after filling
print("Missing values after filling:")
print(df.isnull().sum())
```

Cell 8 — Standardize binary yes/no columns to integers (0 or 1)

Explanation: - Columns like `schoolsup`, `famsup`, `paid`, `activities`, `nursery`, `higher`, `internet`, `romantic` often use `yes` / `no` strings. - We convert them to integers `1` (yes) and `0` (no) to make them easy to query and store.

```
binary_cols =
['schoolsup','famsup','paid','activities','nursery','higher','internet','romantic']

for col in binary_cols:
    if col in df.columns:
        df[col] = df[col].astype(str).str.lower().map({'yes': 1, 'no': 0})
        # If mapping produced NaN (unexpected values), fill with 0
        df[col] = df[col].fillna(0).astype(int)

# Show a small sample
if any(c in df.columns for c in binary_cols):
    display_cols = [c for c in binary_cols if c in df.columns]
    print(df[display_cols].head())
```

Cell 9 — Make some categorical values more readable

Explanation: - Map short codes to readable text for `sex`, `address`, `famsize`, and `pstatus`. - This helps when you present results (easier to understand on social media or in reports).

```
# Mapping examples (only applied if the column exists)
if 'sex' in df.columns:
    df['sex'] = df['sex'].map({'M': 'Male', 'F': 'Female'}).fillna(df['sex'])

if 'address' in df.columns:
    df['address'] = df['address'].map({'U': 'Urban', 'R':
'Rural'}).fillna(df['address'])

if 'famsize' in df.columns:
    df['famsize'] = df['famsize'].map({'LE3': '<=3', 'GT3':
'>3'}).fillna(df['famsize'])

if 'pstatus' in df.columns:
    df['pstatus'] = df['pstatus'].map({'T': 'Together', 'A':
'Apart'}).fillna(df['pstatus'])

# Show a few rows to confirm
cols_to_show = [c for c in ['sex','address','famsize','pstatus'] if c in
df.columns]
if cols_to_show:
    print(df[cols_to_show].head())
```

Cell 10 — Create derived features

Explanation: - Create `avg_grade` as the average of `G1`, `G2`, `G3`. - Create `passed` as a simple label: `1` if final grade `G3` >= 10, otherwise `0`. - These are useful for analysis or as simple targets for models.

```
grade_cols = [c for c in ['g1','g2','g3'] if c in df.columns]
if grade_cols:
    df['avg_grade'] = df[grade_cols].mean(axis=1)
    # final passing rule (change threshold if your institution uses a different
pass mark)
    if 'g3' in df.columns:
        df['passed'] = (df['g3'] >= 10).astype(int)

# Show the new columns
print(df[['avg_grade','passed']].head())
```

Cell 11 — Final checks before saving/loading

Explanation: - Quick sanity checks: grade ranges and absence non-negativity. - Print data types and a short statistical summary for numeric columns.

```
# Sanity checks
if 'g1' in df.columns:
    assert df['g1'].between(0,20).all(), "G1 has values outside expected 0-20
range"
if 'g2' in df.columns:
    assert df['g2'].between(0,20).all(), "G2 has values outside expected 0-20
range"
if 'g3' in df.columns:
    assert df['g3'].between(0,20).all(), "G3 has values outside expected 0-20
range"
if 'absences' in df.columns:
    assert (df['absences'] >= 0).all(), "Absences contains negative values"

# Dtypes and numeric summary
print(df.dtypes)
print(df.describe().T)
```

Cell 12 — Save a cleaned CSV copy (optional)

Explanation: - Save the cleaned DataFrame to `student_cleaned.csv` as a backup before loading into the database. - This file is safe to attach or share with collaborators.

```
clean_csv_path = 'student_cleaned.csv'
df.to_csv(clean_csv_path, index=False)
print(f"Saved cleaned data to {clean_csv_path}")
```

Cell 13 — Prepare PostgreSQL connection credentials (safe way)

Explanation: - We will store sensitive credentials in a `.env` file and load them with `python-dotenv`. - Example `.env` content (create a file named `.env` next to your notebook):

```
DB_USER=your_db_user
DB_PASS=your_db_password
DB_HOST=localhost
DB_PORT=5432
DB_NAME=your_database_name
```

- The code below reads these variables and builds a SQLAlchemy connection string.

```
# Load database credentials from environment variables (.env must exist)
db_user = os.getenv('DB_USER')
db_pass = os.getenv('DB_PASS')
db_host = os.getenv('DB_HOST', 'localhost')
db_port = os.getenv('DB_PORT', '5432')
db_name = os.getenv('DB_NAME')

if not all([db_user, db_pass, db_name]):
    raise ValueError("Please set DB_USER, DB_PASS and DB_NAME in your .env file
before continuing.")

# Create SQLAlchemy engine URL
engine_url = f"postgresql+psycopg2://{db_user}:{db_pass}@{db_host}:{db_port}/
{db_name}"
engine = create_engine(engine_url)

print("Engine created for:", engine_url)
```

Cell 14 — Prepare SQL table schema mapping and load into Postgres

Explanation: - We provide a `dtype` mapping from DataFrame column names to SQL types using SQLAlchemy `types`. - `df.to_sql(..., if_exists='replace')` will create the table and insert the cleaned rows. - Use `if_exists='append'` after the first load to keep adding new batches.

```
# Build a dtype mapping. Adjust sizes as you like (e.g., VARCHAR length)
sql_types = {}

# Example mappings: strings to VARCHAR, integers to INTEGER, floats to FLOAT
for col in df.columns:
    if df[col].dtype == 'int64':
        sql_types[col] = types.INTEGER()
    elif df[col].dtype == 'float64':
        sql_types[col] = types.FLOAT()
    else:
        # Default for other columns (object, category): use VARCHAR
        # Use a sensible max length (here 100). Adjust if you expect longer
text.
        sql_types[col] = types.VARCHAR(length=100)

# Load into Postgres. This will create or replace the table named 'students'.
try:
    df.to_sql('students', con=engine, if_exists='replace', index=False,
dtype=sql_types)
    print("Loaded cleaned data into table 'students' in Postgres.")
```

```
except Exception as e:
    print("Error loading data into Postgres:", e)
```

Cell 15 — Verify data in Postgres (quick queries)

Explanation: - Run simple SQL queries to confirm the load worked. - We run a row count and show a few sample rows.

```
from sqlalchemy import text

with engine.connect() as conn:
    try:
        result = conn.execute(text("SELECT COUNT(*) FROM students;"))
        count = result.fetchone()[0]
        print(f"Rows in students table: {count}")

        # Read the first 5 rows with pandas (convenient for display)
        sample_df = pd.read_sql_query(text("SELECT * FROM students LIMIT 5;"),
con=engine)
        display(sample_df)
    except Exception as e:
        print("Error querying Postgres:", e)
```

Cell 16 — Wrap the process into a reusable function

Explanation: - Create a `run_etl` function that accepts a CSV path and an engine and runs the pipeline. - This helps automate repeating the process for new datasets or scheduling.

```
def run_etl(csv_path: str, engine):
    """Run the ETL steps: load CSV, clean, and write to Postgres.
    This is a simplified wrapper that uses the logic above.
    """
    df_local = pd.read_csv(csv_path)

    # Basic normalization of column names
    df_local.columns = df_local.columns.str.strip().str.lower().str.replace(' ',
'_')

    # Convert numeric-like columns (example list from above)
    for col in num_cols:
        if col in df_local.columns:
            df_local[col] = pd.to_numeric(df_local[col], errors='coerce')
```

```
    # Fill missing numeric values with medians
    existing_num = [c for c in num_cols if c in df_local.columns]
    for c in existing_num:
        df_local[c] = df_local[c].fillna(df_local[c].median())

    # Fill other columns with mode or placeholder
    for c in df_local.columns:
        if df_local[c].isnull().any():
            try:
                df_local[c] = df_local[c].fillna(df_local[c].mode().iloc[0])
            except Exception:
                df_local[c] = df_local[c].fillna('Unknown')

    # Simple binary mapping example
    for col in binary_cols:
        if col in df_local.columns:
            df_local[col] = df_local[col].astype(str).str.lower().map({'yes':
1,'no':0}).fillna(0).astype(int)

    # Derived columns
    if all(c in df_local.columns for c in ['g1','g2','g3']):
        df_local['avg_grade'] = df_local[['g1','g2','g3']].mean(axis=1)
        df_local['passed'] = (df_local['g3'] >= 10).astype(int)

    # Write to Postgres (replace table)
    df_local.to_sql('students', con=engine, if_exists='replace', index=False)
    print("ETL completed: data loaded into 'students'.")

# Example usage:
# run_etl('student.csv', engine)
```

Cell 17 — Notes, next steps and sharing tips (short)

Explanation / Tips for posting: - When you post on social media, show 3 things: a short problem statement, a before/after data sample, and the final SQL query or screenshot of the DB table. - Export the notebook as HTML or PDF from Jupyter (`File -> Download as -> HTML / PDF`) for a clean share. - Add a `README.md` explaining how to run the notebook and how to set up the `.env` file. - If you want, also add a small SQL file with the `CREATE TABLE` statement for people who do not use pandas.

End of notebook.

If you want, I can: - export this content as a real `.ipynb` file you can download, or - make a GitHub-ready repo (README, notebook, `.env.example`, sample CSV), or - shorten the notebook to a 1-page tutorial image for posting on social media.

Tell me which of the three you prefer and I will create it next.