SLEEP_QUALITY_PREDICTION

📌 1. IMPORT LIBRARIES

What this part does

You import:

- **pandas, numpy** → for data handling and numerical ops
- **matplotlib, seaborn** → for plotting and visualizations
- **warnings** → to suppress unnecessary warnings
- **sklearn.model_selection** → `train_test_split`, `cross_val_score`, `GridSearchCV`
- **sklearn.preprocessing** → `StandardScaler`
- **sklearn.metrics** → `r2_score`, `mean_squared_error`, `mean_absolute_error`
- **Models:**
    - LinearRegression, Ridge
    - RandomForestRegressor
    - GradientBoostingRegressor
    - VotingRegressor
    - KNeighborsRegressor
    - SVR
    - XGBRegressor
- **permutation_importance** → model-agnostic feature importance
- Colab tools: `files`, `io`

Why it's important

- These are the **core tools** for building, training, comparing, and explaining your models.
- You're mixing:
    - **Simple linear models**,
    - **Tree-based ensemble models**,
    - **Boosting models**,
    - **Distance-based (KNN)**,
    - **Kernel-based (SVR)**

This variety is important because **no single algorithm is best in theory** → you benchmark multiple and then choose.

📌 2. LOAD DATASET

What the code does

- Uses `files.upload()` (Google Colab) to upload a CSV.
- Reads it with `pd.read_csv(io.BytesIO(uploaded[filename]))`.
- Prints shape and displays `.head()`.

Why it's important

- Ensures the **correct file** is loaded.
- `.head()` gives a **quick sanity check**:
    - Are column names correct?
    - Is `Sleep_Quality_Score` present?
    - Are there weird NaNs or strings?

If anything is wrong, this is where you catch it first.

📌 3. DETECT CAFFEINE COLUMN

What the code does

You loop over `df.columns` and check:

```
if "caff" in c.lower() or "caffeine" in c.lower():
    caff_col = c
```

So it automatically finds a column like `"Caffeine_Intake_mg"`.

Why this is important

Later, you use **caffeine** in a special way:

- You know from feature importance that **Caffeine Intake is dominating** the model.
- You want to **reduce its dominance** but not delete it entirely.
- To do that, you implement **Strategy A: Sample Weighting** based on caffeine.

For that, you need to know **which column is caffeine** without hardcoding the name (so your code still works if the CSV header changes slightly).

So this step makes the entire pipeline **more robust and reusable**.

📌 4. FEATURE DISTRIBUTIONS

What the code does

You create a 3×3 grid of histograms:

- Loops over each column of `df`
- Plots its distribution
- Titles each subplot with the feature name

Why this is important

- Helps you understand:

    - Is the feature **normal, skewed, or bimodal**?
    - Are there **outliers or extreme values**?
- You can visually guess:
    - Whether scaling is needed
    - Whether log transforms might help
    - If some variables have almost constant values

This ties into model performance because:

- **Tree-based models** handle unscaled features fine, but **SVR, KNN, Linear models** are affected by scale and outliers.
- Knowing distributions helps in making **preprocessing decisions**.

📌 5. CORRELATION HEATMAP

What the code does

You compute `df.corr()` and plot a clustered heatmap.

Why this is important

- It tells you **how strongly each feature correlates** with:
    - The target: `Sleep_Quality_Score`
    - Other features: multicollinearity

Key outcomes:

- If **Caffeine Intake** has a very high correlation with sleep quality, that explains:
    - Why tree-based models (and GBR) give it high feature importance.
    - Why the model starts to rely heavily on caffeine.
- You later **address this dominance** using:

    - Permutation importance
    - Weighting strategies

So this step **diagnoses the problem** you fix later.

📌 6. TRAIN-TEST SPLIT + SCALING

What the code does

- Defines `X` = all features except `Sleep_Quality_Score`
- `y` = `Sleep_Quality_Score`
- Splits data: `80% train / 20% test` via `train_test_split`.
- Applies `StandardScaler` to `X_train` and `X_test`.

Why this is important

- **Train-test split**:
    - Ensures you evaluate performance on **unseen data** (test set).
    - Prevents **data leakage** (training and testing on same data).
- **Scaling**:
    - Many models **expect normalized input**, especially:
        - Linear regression, Ridge
        - SVR
        - KNN
    - Without scaling, features with larger numeric scale dominate distances and gradients.

Even though tree-based models (RandomForest, GBR, XGB) **don't need scaling**, you're training **all models on the same scaled version** for consistency.

📌 7. BASELINE MODEL TRAINING & COMPARISON

What the code does

You train 7 models:

- Linear Regression
- Ridge Regression
- Random Forest
- XGBoost
- KNN
- SVR
- Gradient Boosting

For each model you:

- Train on `X_train_scaled, y_train`
- Predict on:
    - Train → `y_pred_train`
    - Test → `y_pred_test`
- Compute:
    - **Train R², Test R²** → goodness of fit
    - **Train RMSE, Test RMSE** → error scale
    - **Train MAE, Test MAE** → average absolute error
    - **CV mean/stdev (5-fold cross-validation R²)** → stability

Then you build `comparison_df` and **sort by Test R²**.

Why this is important

- This is your **baseline benchmarking**.

- It answers:
    - Which family of models works best on this problem?
    - Is the model overfitting? (high train R^2, low test R^2)
    - Which model generalizes best?

Suppose **Gradient Boosting** ends up on top → that's why you:

- Choose it as your **main model family**
- Tune it further
- Enhance it with sample weighting

THIS step ensures you **don't blindly pick a model**.

📌 8. VOTING REGRESSOR (ENSEMBLE)

What the code does

You build a `VotingRegressor` with:

- GBR
- XGB
- RandomForest

Then:

- Fit on `X_train_scaled, y_train`
- Predict `y_pred_voting`
- Compute R^2, RMSE, MAE.

Why this is important

A VotingRegressor averages predictions from several strong models:

$$
\hat{y} = \frac{y_\text{GBR} + y_\text{XGB} + y_\text{RF}}{3}
$$

Benefits:

- **Reduces variance** → if one model overfits a bit in one region, others balance it.
- **More robust** → different models make different kinds of mistakes.
- Sometimes yields **higher R² and lower error** than any single model.

You later compare this to:

- Best single baseline model
- Weighted + tuned models

So this gives you a **strong ensemble benchmark**.

📌 9. BASELINE GBR (UNWEIGHTED) + TUNING

What the code does

1. **Baseline unweighted GBR**:
    - Train `GradientBoostingRegressor(random_state=42)` on **unscaled `X_train`**.
    - Predict on `X_test`.
    - Evaluate R², RMSE, MAE.
2. **Hyperparameter tuning**:
    - Use `GridSearchCV` on:
        - `n_estimators`
        - `learning_rate`
        - `max_depth`

        - `subsample`
    - 5-fold CV.
    - Fit on `X_train`.
    - Get `best_gbr = grid_gbr.best_estimator_`.
    - Predict `pred_tuned`.
    - Evaluate metrics again.

Why this is important

- Baseline GBR tells you: **how good is plain, default Gradient Boosting**.
- The tuned GBR improves:
    - **Bias-variance tradeoff** (depth + n_estimators + learning rate).
    - **Control overfitting** via `subsample`.
- Comparing:
    - Baseline GBR vs Tuned GBR
      answers: *"Does tuning significantly help?"*

📌 10. CLEAN VISUAL DIAGNOSTICS (TUNED GBR)

You create several plots for the tuned GBR (unweighted):

1. **Correlation matrix** → Already seen, but repeated here in diagnostics.
2. **Pairplot** → Relationship between numeric features.
3. **Feature importance** (`best_gbr.feature_importances_`)
    - Shows which features GBR uses most.
4. **Actual vs Predicted plot**
    - Diagonal alignment = good predictions.
5. **Residual distribution**

    - Should be centered around 0.

6. **Residuals vs Predicted**
    - Checks for patterns (if residuals show structure, model may miss something).
7. **Boxplot (outliers)**
    - Detects extreme values.

Why this is important

This is your **model diagnostic section**:

- Checks whether the model:
    - Is biased (systematic under-/over-prediction)
    - Handles variance well
    - Suffers from outliers or skew
- Helps answer questions like:
    - *"Is this model trustworthy?"*
    - *"Can we use it for real-world prediction?"*

📌 11. PERMUTATION IMPORTANCE (VOTING MODEL)

What the code does

You compute `permutation_importance(voting_model, X_test_scaled, y_test, ...)`.

- For each feature:
    - Randomly shuffle that feature column.
    - See how much model performance drops.
    - Large drop = feature very important.
- You sort these results and plot them.

- You specifically print caffeine's importance: `perm_ser[caff_col]`.

Why this is important

- **Permutation importance is model-agnostic** and more reliable than raw `.feature_importances_`.
- It **confirms**:
    - Whether caffeine truly dominates the predictive power.
- This is the **evidence** that justifies Strategy A.

So this step answers:

> "Is caffeine really too powerful in this model?"

If yes → we move to sample weighting.

📌 12. STRATEGY A — SAMPLE WEIGHTED GBR (CAFFEINE-AWARE)

What the code does

1. Extract caffeine values from **training set only**:

```
c_vals = X_train[caff_col].astype(float).values
```

2. Normalize them to [0,1]:

```
c_norm = (c_vals - c_vals.min()) / (c_vals.max() - c_vals.min() + 1e-9)
```

3. Define **sample weights**:

```
alpha = 2.0
sample_weights = 1 / (1 + alpha * c_norm)
```

- High caffeine → larger `c_norm` → smaller weight.

- Low caffeine → larger weight.

4. Train:

```
gbr_sw = GradientBoostingRegressor(random_state=42)
gbr_sw.fit(X_train_scaled, y_train, sample_weight=sample_weights)
```

5. Predict & evaluate on test data.

Why this is important

This is the **fairness / dominance control step**.

- You **don't drop caffeine**, which would lose information.
- Instead, you **reduce its influence** by:
    - Making high-caffeine samples less important in training.
- This encourages model to:
    - Use **other features** more.
    - Not overfit to caffeine-heavy regions.

Effectively, you're saying:

> "Caffeine is useful, but don't let it dictate everything."

This improves:

- **Fairness** (sleep quality not only blamed on caffeine)
- **Robustness** (if caffeine measurements are noisy, model is less fragile)

📌 13. STRATEGY A + TUNED GBR (Sample-weighted Tuning)

What the code does

You now do **GridSearchCV**, but this time:

- On a `GradientBoostingRegressor`
- Using the **same sample_weight** (downweighting high-caffeine) during `.fit()`:

grid_sw.fit(X_train_scaled, y_train, sample_weight=sample_weights)

You search over:

- `n_estimators`
- `learning_rate`
- `max_depth`
- `subsample`
- `max_features`

Then:

- Get `best_sw_tuned = grid_sw.best_estimator_`
- Predict: `y_pred_sw_tuned`
- Evaluate metrics.

Why this is important

Here you combine:

1. **Better training objective** → Weighted samples (fairer)
2. **Better model capacity** → Tuned hyperparameters

So this becomes your **most advanced, refined, and fair model**.

If its:

- **R^2 is highest**
- **RMSE & MAE are among lowest**

→ This is your **final winner model**.

📌 14. FINAL COMPARISON TABLE (ALL APPROACHES)

What the code does

You build `final_comp` containing metrics for:

- Best Single Baseline (by Test R^2)
- Voting Ensemble
- Baseline GBR (Unweighted)
- Tuned GBR (Unweighted)
- GBR (Sample-Weighted)
- GBR (Sample-Weighted + Tuned)

And then:

- Print table
- Plot bar chart of R^2 for visual comparison.

Why this is important

This answers the big question:

> "After all this complexity, **which one is actually the best?**"

It makes it easy to argue:

- We started with multiple baselines.
- We built an ensemble.
- We tried plain and tuned boosting.
- We then made it **caffeine-aware and fair**.
- The final **Sample-Weighted + Tuned GBR** wins overall.

📌 15. MODEL LEADERBOARD (Ranked by R^2)

What the code does

You:

- Sort `final_comp` by `r2` descending.
- Print a leaderboard:

1. GBR (Sample-Weighted + Tuned) — R^2 = ...
2. Voting Ensemble — R^2 = ...
3. Tuned GBR — R^2 = ...
...

- Plot bar chart (R^2 vs model names).

Why this is important

This is your **storytelling and presentation layer**.

- Anyone reading your notebook / report can see:
    - **Who is the champion model**.
    - **How much better it is** than the others.
- Very useful for:
    - Teachers / viva panels
    - Project evaluators
    - Future you (when you revisit this project)

🧩 Big Picture: How Every Part Helped Improve the Model

Let's summarize tightly:

1. **EDA (distributions, heatmap)**
   → Helped find patterns, skewness, and caffeine dominance.

2. **Multiple baseline models**
   → Ensured you **chose the right model family** (Gradient Boosting) by evidence.

3. **Voting Regressor**
   → Showed the benefits of **ensembles** over single models.

4. **Unweighted GBR + tuning**
   → Extracted maximum power from Gradient Boosting alone.

5. **Permutation importance (voting)**
   → Confirmed that **caffeine was over-dominant**.

6. **Strategy A (sample-weighted GBR)**
   → Directly addressed **caffeine dominance** by reducing its training influence.

7. **Strategy A + Tuning**
   → Combined **fairness + optimization** to get:

   - Best predictive performance
   - Better feature balance
   - Conceptually fair model

8. **Final comparison + leaderboard**
   → Clearly shows **why final model is chosen**, not just by intuition but by metrics.
