**CODE REVIEW – EXECUTION HELPFUL HINTS – OUTPUT INTERPRETATION**

**Code Organization Review**

1. *Argument Parsing (parse\_args)*  
   Collects runtime choices: I/O paths, CV settings, grid caps, and flags to include/exclude transformation families.
2. *Data I/O & Data Typing (read\_data, identify\_column\_types)*Loads CSV/XLSX, then, for each target, it drops other targets and splits predictors into categorical vs numeric for upcoming feature engineering transformations.
3. *Transformation Menus (*Reminder: in the context of data science, *transformation menus* refer to the various features of engineering techniques and methods used to modify, clean, and prepare the raw data for analysis, typically, using Machine Learning algorithms*)*
   * Categorical (categorical\_options): builds options from flags: *one-hot*, *one-hot (rare grouped)*, *OOF target encoding*, *smoothed target encoding*.
   * Numeric (numeric\_options): builds options: *raw*, *raw+scaler(s)*, *imputer(s)*, *imputed+scaler(s)*.  
     Family-disable flags prune these menus to control the explosive size of the cross-product of predictor-transformation combinations (relevant to the grid definition described below).
4. *Grid construction (build\_transform\_grid\_for\_combo)*For each (size ≥2) base predictor combination (combo), this function forms the complete Cartesian product of all transformation choices that can be applied to combo's predictors with the predictor(s) and calls each element of the Cartesian product a "grid". In other words, all the grids in the same combo summarize the possible combinations between the base predictor combination and their applicable transformations.
5. *Virtual Memory-supported design matrix (create\_memmap)*Streams each transformed block to a *numpy.memmap* matrix on the disk for each grid. Any column NaNs are mean-filled per column to keep statistics finite.
6. *Optional in-place z-scoring (standardize\_memmap\_inplace)*Two-pass block algorithm computes means/stds in chunks, and then it writes back the in the memmap to make elastic-net penalties comparable across heterogeneous features.
7. *Overall metric via nested CV (cv\_elasticnet\_r2)*
   * Outer K-fold, i.e., out-of-sample estimates R2
   * Inner K-fold (ElasticNetCV) tunes *α* and *l1\_ratio* on the training fold only.  
     Returns mean±std R2 across outer folds along with the average-tuned hyperparameters.
8. *Results*  
   For each target/grid, the script records and outputs to a csv file: *target, combo\_id, grid\_id, base\_columns, variant, n\_rows, n\_cols, r2\_mean, r2\_std, alpha, l1\_ratio*.

**Analysis: Statistics & Math Under the Hood**

* *Elastic Net Algorithm Objective*

Given features , the Elastic Net solves

where α>0 controls the overall penalty and ρ∈[0,1], also called *l1\_ratio* balances ridge (ρ=0) and lasso (ρ=1) in ENet.  
We chose ENet to evaluate multi-predictor-target combinations because it effectively handles multicollinearity (arising from numerous one-hot dummies compared to more compliant target encodings) and can select/shrink features in very wide grids (review the definition of what a *grid* is).

* *Target encodings*
* *OOF* target encoding: on each training fold, map category c to

; apply to validation fold to avoid leakage.

* *Smoothed target encoding:* (James–Stein shrinkage), stabilizing rare levels
* *Mean imputation & standardization*
* *Mean imputation* keeps Pearson/least-squares well-defined; imputed entries have zero-centered residual, avoiding artificial covariance.
* *Z-scoring* (optional) makes columns unit-scale so α and ρ are comparable across dummies, encodings, and scaled numeric predictors.
* *Nested cross-validation* & R2
* *Outer CV promotes generalization:* for each split, fit on train (with inner tuning), score on held-out test.
* *Performance Metric* is defined as where a negative R2 indicates performance worse than predicting the mean.
* *Inner CV picks values for α* and *ρ* using only the training fold, preventing optimistic bias.

*Rationale:*

Nested CV is based on two loops. The **outer loop** splits the data into only two subsets, the **outer-train** and **outer-test**, i.e., there's no separate "validation" step there. For each outer-train subset, the **inner loop** runs K-fold CV by splitting outer-train into **inner-train**/**inner-val** folds to tune α and ρ hyperparameters. Then it refits on the entire outer-train subset using the chosen hyperparameter settings and evaluates the goodness of the fit once on the unseen **outer-test** fold. Repeat this process across the outer folds and then average the outer-test scores to return the final estimate. Because the outer-test fold never influenced hypermarameter tuning, overfitting is minimized and the estimate is not inflated by it, hence the term "optimistic bias". Had we tuned on the entire dataset or used the test fold during training, we would have leaked information and reported an overly optimistic R2.

**Code Steps in Brief**

1. Choose a target; drop the other targets from predictors.
2. Enumerate all predictor base combos for sizes ≥ 2
3. For each combination, create a menu of transformations for each predictor based on the end user's selected flag values.
4. Take the Cartesian product across predictors to get all grids (with an optional ceiling defined by relevant flags).
5. For each grid:  
   a) Transform each predictor block, mean-fill NaNs, stream into a hard disk memmap.  
   b) Optionally standardize in place, in blocks.  
   c) Run outer K-fold: for each split, run ElasticNetCV on the training fold (inner K-fold), predict held-out test, and collect R2.
6. Aggregate per grid: mean±std R2 and average tuned α and ρ.
7. Write one csv file per target; higher R²\_mean = better overall joint predictive power.

**Tips & tricks for efficient runs**

* Put --memmap\_dir on an SSD and ensure plenty of free space.
* Start conservative: --disable\_cat\_oof --disable\_cat\_smoothed --disable\_num\_imputer\_scalers and raise --min\_count\_threshold.
* Use caps: --max\_combos\_per\_target and --max\_grids\_per\_combo.
* Turn on --standardize for robust tuning and adjust --l1\_ratio\_grid if you expect high sparsity (include 0.9 or 1.0).
* Tune --block\_size to fit available RAM, and increase it to larger values for faster response, provided it fits in RAM.
* Set *--random\_state* for reproducibility and use *--verbose* to monitor progress

**Quick Review on Encoding Techniques**

**Out-of-Fold (OOF) target encoding** computes a category's mean of the target on each training fold. Then it applies it to that fold's hold-out, preventing leakage while turning high-cardinality categories into a single, dense numeric signal. **Smoothed target encoding** shrinks each category mean toward the global mean to stabilize rare levels, typically via

where a larger m ⇒ more shrinkage.

The **Category Encoders** library is a toolbox of encoders (e.g., Target, Leave-One-Out, CatBoost, James-Stein, Helmert, Hashing) that fit on training data and transform categories into numeric representations to reduce dimensionality and leakage while preserving signal. **Minimum Sample Threshold** (rare-level grouping) collapses infrequent categories into an "*Other*" bucket before one-hot/target encoding cuts dummy width, reduces variance caused by tiny groups, and improves robustness for sparse levels.

**Merges rare categories** into a single category called "*Other*" *before* turning categories into numbers, e.g., via one-hot or target encoding. By doing this the script **decreases the number of generated features** (fewer dummy columns in the case of hot encoding), **stabilizes estimates** because tiny groups with almost no data do not fit well in the greater data set (reduses high-variance, quiesces noisy effects), and **reduces overfitting**, making models more robust when very sparse categories are present in the data.

**Execution flags (definitions) Categorized by Task**

* **I/O, file/RAM mngt**
  + --data (required): Network path to csv/xlsx file
  + --sheet: excel sheet name (if raw data source is an xls/xlsx file)
  + --targets: One or more target columns evaluated separately, excluding the other targets
  + --output\_dir: where result csv files are saved per target
  + --print\_output\_dir: the text file where the stdout gets saved
  + --memmap\_dir: folder for virtual memory (hard disk) matrices (use SCSI RAIDs or SSD with enough disk space available)
* **Encoding/scaling hyperparameters**
  + --min\_count\_threshold: groups rare categorical levels into "Other" for the *rare one-hot* option
  + --k\_folds: this hyperparameter is used only for *OOF target encoding*
  + --random\_state: ensures reproducibility of data row shuffling and train-test splitting b/c the random number generator is initialized with the same seed each time the script is run (the seed in this script is 54)
  + --block\_size: column-segment size used for in-place standardization (smaller size means less usable RAM).
  + --standardize: Z-scores all grid features on the memmap before modeling (recommended to make elastic net penalties comparable).
* **Verbosity & workload caps (ceiling flags)**
  + --verbose: print a one-line summary per grid
  + --max\_combos\_per\_target: cap number of base predictor sets (size ≥ 2).
  + --max\_grids\_per\_combo: cap number of transformation grids per base predictor set
* **Elastic Net CV (overall metric)**
  + --cv\_folds: Outer folds for **out-of-sample R2**.
  + --enet\_cv\_folds: Inner folds for **ElasticNetCV** hyperparameter tuning
  + --l1\_ratio\_grid: search values for L1/L2 values experimentation (e.g., 0.1, 0.5, 0.9).
  + --alphas\_grid: (optional); if omitted, ElasticNetCV picks one automatically based on internal logic. Reminder: An alpha grid is a hyperparameter that includes a list of regularization strengths (α values) to test, aiming to find the one that yields the best CV score during a k-fold. Larger α values apply more pressure, directing the ENet coefficients towards a close to zero value (shrinkage). Using this flag will be more rewarding if standardization is enforced on the features.
* **Grid-shrinking switches (turn off encoding/imputation/scaling)**
  + *Categorical*: --disable\_cat\_one\_hot, --disable\_cat\_rare, --disable\_cat\_oof, --disable\_cat\_smoothed.
  + *Numeric*: --disable\_num\_raw, --disable\_num\_raw\_scalers, --disable\_num\_imputers, --disable\_num\_imputer\_scalers.
  + *Fine-grained*: --disable\_scaler\_standard, --disable\_scaler\_minmax, --disable\_scaler\_robust, --disable\_imputer\_mean, --disable\_imputer\_median, --disable\_imputer\_most\_frequent

**Output CSV columns (how to interpret them)**

* target: The target column evaluated for this row.
* combo\_id, grid\_id: IDs are associated with a base predictor set and a specific transformation choice
* base\_columns: Pipe-delimited names of the predictors in the base combo, e.g., col1|col2|col3|….|colN, where N is a positive integer.
* variant: short description of the chosen transformation per predictor, e.g., A=one\_hot | B=oof\_target | C=imp\_mean+standard
* n\_rows, n\_cols: Dataset rows used and number of transformed feature columns in that grid (after feature eng app'n)
* r2\_mean, r2\_std: **Outer-CV Elastic Net R2** mean and std across folds (negative means predicting power is worse than predicting the mean)
* alpha, l1\_ratio: inner fold average-tuned hyperparameters chosen by the inner ElasticNetCV across outer folds (not a refit on complete data).

**Script Execution flag examples**

Below you'll find a series of **copy-paste** command flag examples, grouped by task. Mix and match as needed; the accompanying title hints at what each flag does.

*# --- Minimal run (initiate CSV) ---*

python correlation\_pipeline\_grid\_memmap\_cv\_enet.py \

--data /network/path/to/results.csv \

--print\_output\_dir /network/path/to/print\_output.txt \

--targets "Target1" "Target2"

*# --- Reading Excel with a specific sheet ---*

python correlation\_pipeline\_grid\_memmap\_cv\_enet.py \

--data /path/to/source\_data.xlsx --sheet "Sheet1" \

--print\_output\_dir /network/path/to/print\_output.txt \

--targets "Target1" "Target2"

*# --- Output folders (results + memmap on fast SSD) ---*

python correlation\_pipeline\_grid\_memmap\_cv\_enet.py \

--data /network/path/to/results.csv \

--print\_output\_dir /network/path/to/print\_output.txt \

--output\_dir /fast/out --memmap\_dir /fast/tmp \

--targets "Target1" "Target2"

*# --- Encoding/scaling hyperparams (rare threshold, OOF folds, seed, block size, z-score) ---*

python correlation\_pipeline\_grid\_memmap\_cv\_enet.py \

--data /network/path/to/results.csv \

--print\_output\_dir /network/path/to/print\_output.txt \

--min\_count\_threshold 50 \

--k\_folds 10 \

--random\_state 123 \

--block\_size 2048 \

--standardize \

--targets "Target1"

*# --- Verbose enable + workload ceilings (limit combos explosion) ---*

python correlation\_pipeline\_grid\_memmap\_cv\_enet.py \

--data /network/path/to/results.csv \

--print\_output\_dir /network/path/to/print\_output.txt \

--verbose \

--max\_combos\_per\_target 100 \

--max\_grids\_per\_combo 1000 \

--targets "Target1"

*# --- Elastic Net nested-CV tuning grids ---*

python correlation\_pipeline\_grid\_memmap\_cv\_enet.py \

--data /network/path/to/results.csv \

--print\_output\_dir /network/path/to/print\_output.txt \

--cv\_folds 5 \

--enet\_cv\_folds 5 \

--l1\_ratio\_grid 0.05 0.5 0.95 \

--alphas\_grid 0.0001 0.001 0.01 0.1 1.0 \

--standardize \

--targets "Target1"

*# --- Disable categorical families (keep only one-hot variants) ---*

python correlation\_pipeline\_grid\_memmap\_cv\_enet.py \

--data /network/path/to/results.csv \

--print\_output\_dir /network/path/to/print\_output.txt \

--disable\_cat\_oof --disable\_cat\_smoothed \

--targets "Target1"

*# --- Disable one-hot dummy families (keep only target encoders) ---*

python correlation\_pipeline\_grid\_memmap\_cv\_enet.py \

--data /network/path/to/results.csv \

--print\_output\_dir /network/path/to/print\_output.txt \

--disable\_cat\_one\_hot --disable\_cat\_rare \

--targets "Target1"

*# --- Disable numeric raw+scalers and imputed+scalers (shrink numeric option count) ---*

python correlation\_pipeline\_grid\_memmap\_cv\_enet.py \

--data /network/path/to/results.csv \

--print\_output\_dir /network/path/to/print\_output.txt \

--disable\_num\_raw\_scalers \

--disable\_num\_imputer\_scalers \

--targets "Target1"

*# --- Disable all numeric imputers (use only raw and raw+scalers) ---*

python correlation\_pipeline\_grid\_memmap\_cv\_enet.py \

--data /network/path/to/results.csv \

--print\_output\_dir /network/path/to/print\_output.txt \

--disable\_num\_imputers \

--targets "Target1"

*# --- Disable specific scalers (drop robust; keep standard & minmax) ---*

python correlation\_pipeline\_grid\_memmap\_cv\_enet.py \

--data /network/path/to/results.csv \

--print\_output\_dir /network/path/to/print\_output.txt \

--disable\_scaler\_robust \

--targets "Target1"

*# --- Disable specific imputers (drop most\_frequent & median; keep mean) ---*

python correlation\_pipeline\_grid\_memmap\_cv\_enet.py \

--data /network/path/to/results.csv \

--print\_output\_dir /network/path/to/print\_output.txt \

--disable\_imputer\_most\_frequent --disable\_imputer\_median \

--targets "Target1"

*# --- Practical “production-friendly” run: smaller grid + standardization + caps ---*

python correlation\_pipeline\_grid\_memmap\_cv\_enet.py \

--data /network/path/to/results.csv \

--print\_output\_dir /network/path/to/print\_output.txt \

--targets "Target1" "Target2" \

--output\_dir /fast/out --memmap\_dir /fast/tmp \

--standardize --block\_size 1024 --random\_state 42 \

--disable\_cat\_oof --disable\_cat\_smoothed \

--disable\_num\_imputer\_scalers --disable\_scaler\_robust \

--max\_combos\_per\_target 50 --max\_grids\_per\_combo 500 \

--verbose

**Questions/Answers on Algorithms/Concepts/Statistical Terms used in the Python Script**

Q: What does *data leakage* refer to?

A: Data leakage takes place when information from the validation/test set, or directly from the target, slips into model training, resulting in a model of atypically good performance out of training that fails miserably on unseen data. Common causes (or mistakes) include model fitting during preprocessing with scaling, imputation, encoding on the entire dataset before train/test/validation splitting, or using features that wouldn't be available at the time the model's predictability will be put to the test (see the following Q on temporal leakage). Protect against data leakage by fitting all transformed predictors only on the training fold, using out-of-fold encoders for categoricals, and enforcing proper time-based splits when applicable.

Q: What is *temporal leakage*?

A: Temporal leakage describes the case when future information (relative to the time the model's predictive power is tested) slips into feature training or preprocessing, artificially inflating performance. Examples include the inclusion of predictors recorded after the outcome in the training fold, computing scalers/encoders/aggregates on the entire dataset, using random (non-temporal) cross-validation for time series, or employing "rolling" features that inadvertently carry data beyond the prediction period. Avoid it by using strict time-based splits, fitting all preprocessing only on the training window of time, and utilizing lagged or expanding-window statistics/OOF encoders that never use future data.

Q: What is an *alpha grid*?

A: An alpha grid is the list of candidate regularization strengths α Elastic Net tries during cross-validation to pick the α that yields the best CV score. Larger α drives more coefficients toward zero (more substantial shrinkage), so typically we search for a good α in a log-spaced range, e.g., 1e-4 … 1e2, that works best if features are standardized. ElasticNetCV builds its path if one is not supplied; otherwise, we pass one via alphas\_grid: 0.0001, 0.001, 0.01, 0.1, 1 10

Q: What is the *2-pass block algorithm*?

A: The two-pass block algorithm is a memory-safe way to standardize a huge, disk-backed matrix. In pass 1, the algo streams through the feature matrix by column blocks (i.e, it processes the data sequentially in small chunks or micro-batches avoid loading the entire data set in RAM that's usually limited in size and availability) set by using the flag *size = --block\_size*, computing and storing each block's μ and σ. Next, in pass 2, it streams through the same blocks again and applies in-place (X−μ)/σ (setting σ=1 for zero-std columns). Using blocks keeps peaking RAM space near O(n×block\_size), while the two passes are needed to calculate μ and σ for safe normalization without loading the entire matrix in RAM.

Q: How does a 2-pass algorithm help Elastic Net CV across heterogeneous features?

A: 2-pass algo z-scores every feature (using means/SDs) in a RAM-safe way, so Elastic Net's penalty "observes" heterogeneous features on the same scale, so one-hot dummies, target encodings, and raw numerics don't dominate. That feature makes the α and l1\_ratio search comparable across folds, improving conditioning, convergence, and the stability of CV R2. To avoid leakage, the 2-pass algorithm computes blockwise statistics on training folds only, rather than on the entire dataset.

Q: Can you elaborate as to why *Inner CV picks α, ρ using only the training fold, averting optimistic bias*?

A: In nested cross-validation, the inner CV loop searches hyperparameters (here α and ρ or l1\_ratio) using only the training fold of the outer split. After picking the best α, ρ from inner CV, it refits on the outer-train data using those settings and scores once on the untouched outer-test fold. Because the test fold never influenced hyperparameter tuning, the estimate isn't inflated by overfitting ("optimistic bias"). Tuning on the complete dataset, or peeking at the test fold, would leak information, potentially leading to overly optimistic R2 reports.

Q: What does flag *max\_combos\_per\_target* do?

A: *--max\_combos\_per\_target* caps how many base predictor combinations (of size ≥ 2) the script evaluates for each target. After generating the predictor combos, it keeps only the first N (in the script's generation order) to limit runtime and disk I/O. Per-combo grid expansion by transformation combinations on top of the predictor combinations remains unchanged (use *--max\_grids\_per\_combo* to cap those).

Q: Example of how the flag *max\_combos\_per\_target* is used.

A: This limits the script to the first 100 base predictor combinations per target (size ≥ 2). The per-combo transformation grids will still fully expand to their max number, unless *--max\_grids\_per\_combo* is also throttled to a specific value.

Q: Assume that *--max\_combos\_per\_target* = 100 and *--max\_grids\_per\_combo* = 100 when the script gets set up for execution, but the total number of resultant grids comes to be only 7,300, why's that?

A: The two flags mentioned in the Q are switches to throttle the number of those two types of combinations, but they do not directly control the total number of grids per se. The following equation calculates the script's total number of grids:

where *--max\_grid\_per\_combo* is a script (already explained) flag that puts a value ceiling to the total number of transform options per combo of predictors to be taken into account, *mp* is the number of enabled (dependent upon what flags have been enabled at execution kick off) transform options for predictor *p*, and

*Valid Combos* = *minimum of (all possible predictor combos per target, --max\_combos\_per\_target)*

The total number of grids will drop below the expected 100x100 = 10K when:

There are fewer than 100 valid base predictor combinations

Some valid base predictor combos have transform options

A valid base combo is skipped because >=1 predictor(s) has(ve) zero enabled options based on the flag selected as enabled, which means that

Enabling the optional *--verbose* flag and printing *stdout* to a text file, using the required flag *--print\_output\_dir* (that sets the network path of the text file where all print out on the screen will be saved), will also include the breakdown of predictor combos and their transform grids to help the script user understand how the estimation of the final number of grids takes place in *\_estimate\_total\_grids* helper function.

Q: What is *nested cross-validation*?

A: Nested Cross-Validation (NCV) uses two loops, an outer and an inner loop.

During CV in the outer loop (*outer CV*), the **outer split** creates only an **outer-train** and **outer-test** fold, and skips an outer validation fold.

The *inner CV* runs only on **the outer-train** fold, re-splitting it into inner-train/inner-validation folds to tune the hyperparameters for the pipeline. After selecting the best settings, such as alpha and l1\_ratio for Elastic Net, and making a scaler choice or deciding on an imputer strategy, inner CV refits the full pipeline on the entire **outer-train** and estimates the fit performance score on the untouched **outer-test** fold only once. Then, it averages the outer-test scores collected across the outer CV folds (defined by the flag --cv\_folds) to obtain the final unbiased generalized hyperparameter estimate. The above methodology ensures that all preprocessing/feature selection tuning is fitted within the inner-CV pipeline, separate from the model's evaluation, to prevent data leakage and reduce optimistic bias.

Q: Elaborate on the *"Other" bucket* that cuts dummy width in one-hot/target encoding, reduces variance caused by tiny groups, and improves robustness for sparse levels.

A: That means the script merges rare categories into a single one called "Other" before turning categories into numbers using one-hot or target encoding. This merging decreases the number of extra generated features (fewer dummy columns), stabilizes estimates because tiny groups with almost no data don't fit well (therefore, it reduces high-variance and any noisy effects), and throttles overfitting, making models more robust when categories are very sparse.

Q: Explain how *memmap* works.

A: *numpy.memmap* creates an interface that maps the data stored in a file to an array on disk inside a process's virtual address space, so the OS dynamically loads ("memory-map paging in") only the touched pages of the file into physical RAM, as the array is accessed. Similarly, any modifications are written back to disk by the OS. This on-demand paging mechanism enables working with datasets larger than the available RAM, sharing data across processes, and offering the best performance for **sequential, blockwise** reads/writes.

The script uses *call .flush()* (or close) to persist changes. Potential caveats to memmap and similar disk-backed techniques include the sluggishness of random smaller I/O loads. Additionally, the values' dtype/shape are fixed, whereas making them dynamic requires re-creating the map, which cannot happen due to concurrent writes needing external locking. Performance also depends heavily on fast disk types, such as RAID-based SCSIs or SSDs.

Q: Explain how memmap writes chosen transform blocks to a disk-backed matrix and mean-fills NaNs.

A: Each selected feature transform (e.g., one-hot block, encoded column, scaled numeric) is appended directly into a disk-backed array (numpy.memmap) instead of first concatenating everything in RAM. During that write, any missing values in each block are replaced with the block's column mean, so stats/math logic, such as standardization, Elastic Net, CV, encounter finite data.

Q: Why may we have NaNs during write in the memmap process?

A: NaNs can already exist in your raw numerics, and in some cases, transformations may introduce them, e.g., an OOF/target-encoding lookup for a category that never appears in the training fold, or a numeric column that's entirely missing, so its mean/scale becomes NaN. To ensure all math/stats logic encounters only finite data, create\_memmap mean-fills each block's columns during a *write*.

Q: Do all the imputation, encoding of categorical predictors, and scaling of numeric predictors take place during the memmap process?

A: Each predictor's encoding/imputation/scaling just-in-time computation takes place inside a small pandas block, which gets transformed and streamed into the memmap. During that "*write",* mean-fill any residual NaNs also occurs. After assembling all the blocks, standardization is the only transformation triggered by the optional flag --standardize, which enables a second pass to z-score the entire memmap in place, block by block.

Q: What transformations are covered by *imputed+scalers*?

A: "imputed+scalers" can be interpreted as follows: for each numeric predictor, the script, at first, imputes any missing values (using one of: mean, median, most\_frequent) and then it scales the imputed series (using one of: standard, minmax, robust). By default, that yields 3×3 = 9 variants per numeric column: (mean→standard), (mean→minmax), (mean→robust), (median→standard), …, (most\_frequent→robust). This set of variants will decrease by turning off any imputers or scalers using the appropriate flags, e.g., --disable\_imputer\_median, --disable\_scaler\_robust, or --disable\_num\_imputer\_scalers.

Q: What's the definition of the word *family* in this analysis?

A: A family is a group of transformation options, or transforms, with a similar objective. For example, categorical families include one-hot, rare one-hot, OOF target, and smoothed target; numeric families include raw, raw+scalers, imputers, and imputed+scalers (with scalers and imputers as subfamilies). It's helpful to remember that disabling a family removes all its variants from the grid. Also, the code chooses one option at a time from the applicable family of transformations when forming each grid for each predictor.

Q: If I turn off the imputed+scalers using the proper flag is it safe to assume that the code will apply each imputation method and each scaling method individually to all numeric fields as well as use the numeric fields raw, but not combining any of the imputation or scaling in any way when it identifies which transformations to apply to the predictors?

A: *--disable\_num\_imputer\_scalers* removes only the combined variants (imputer+scaler). You'll still get raw, each imputer alone (imp\_mean, imp\_median, imp\_most\_frequent), and each scaler applied to the raw column (raw+standard|minmax|robust). No imputer+scaler pairs will be created. To suppress scaling-only, add the flag *--disable\_num\_raw\_scalers*. The optional global flag *--standardize* (that performs post-assembly z-scoring) is independent of all above transformations.

Q: What does *finite stats* mean?

A: "*finite stats*" means that (transformed) predictor values in the calculations are real numbers (not NaN, +∞, or −∞), so critical mathematical quantities like means, variances, correlations, etc. can be safely calculated. In this script's pipeline, mean-imputing missing entries and safeguarding against zero standard deviations prevent NaN propagation and keep all calculations computationally feasible.

Q: What does *r2\_mean* or *variance explained out-of-sample* mean?

A: r2\_mean is the average 𝑅2 (see eq. below) scored on the outer-CV folds, so it estimates how much of the target's variance the model explains out-of-sample.

where 1 is an ideal prediction, 0 matches the baseline mean, and negative values signal predictions worse than the baseline. In a nutshell, higher r2\_mean ⇒ expect a model with stronger generalizability.