Stratified, Motif-Unique Sampler — Step-by-Step Tutorial

This section explains just the stratification & sampling code you shared. It covers **what each line does**, **the order of operations**, and **why** each choice is made. You can paste this into a README/tutorial.

1) High-level goal

Build a **balanced, diverse** subset of probes across binding strengths by:

- **Stratifying** on log-intensity (even coverage across the range), and
- **Enforcing motif uniqueness** (avoid near-duplicate sequences), with **deterministic randomness** for reproducibility.

2) Inputs & initial split by class

```
seeds = range(10000)

unbound_probes = gcPBM_myc_final[gcPBM_myc_final['Log Intensity'] <= 8]
weak_probes    = gcPBM_myc_final[(gcPBM_myc_final['Log Intensity'] >  8) &
(gcPBM_myc_final['Log Intensity'] < 9)]
strong_probes  = gcPBM_myc_final[gcPBM_myc_final['Log Intensity'] >= 9]
```

What happens:

- Creates a long list of integer seeds for per-pick reproducibility.
- Partitions the full table into three non-overlapping subsets using **fixed log-intensity thresholds**: ≤8 (unbound), 8–9 (weak), ≥9 (strong).

Why:

- We'll sample **within** each class to keep the final dataset balanced.
- Using explicit thresholds ties the classification to the empirical distribution you chose earlier.

3) The sampler's contract

```
def sample_probes(df, num_samples):
    """
    Evenly sample across 0.1-wide log-intensity bins, forbidding repeated
motifs.
    Rotate tie-breaker motif keys (6mer→8mer→10mer→12mer) if a bin runs
out.
    Random draws use `seeds[i]` so identical code + data ⇒ identical sample.
    """
```

What: A reusable function that takes a **single class subset** and returns `num_samples` rows satisfying stratification + uniqueness.

Why: Encapsulates the logic so you can call it for unbound/weak/strong the same way.

4) Motif keys, sample container

```
motif_col = iter(['6mer', '8mer', '10mer', '12mer'])
samples   = []
```

What:

- `motif_col` is a **rotating iterator** over k-mer columns used to enforce uniqueness when a given key can't find new unique samples.
- `samples` accumulates picked rows (each as a 1-row DataFrame).

Why:

- Prioritizing different k-mers protects against degenerate repeats (e.g., same 8-mer but different 6-mers) and encourages diversity.

5) Build 0.1-wide intensity bins

```
bins = np.arange(round(df['Log Intensity'].min(), 1),
                 round(df['Log Intensity'].max(), 1) + 0.1, 0.1)
df = df.copy()
df['bin'] = pd.cut(df['Log Intensity'], bins)
```

What:

- Computes bin edges every 0.1 across the class-specific range.
- Labels each row with a categorical `bin`.

Why:

- **Stratification**: iterating bins distributes picks across the full intensity spectrum (not just dense regions).

6) Main selection loop

```
i = 0
motif = next(motif_col)
while i < num_samples:
    len_motifs = []
    for br in df['bin'].cat.categories:
        group   = df[df['bin'] == br]
        uniques = group.loc[~group[motif].isin(
            pd.concat(samples)[motif] if samples else []
        ), motif].unique()
        len_motifs.append(len(uniques))
        if len(uniques):
            pick = group[group[motif].isin(uniques)].sample(n=1,
```

```
random_state=seeds[i])
            samples.append(pick)
            i += 1
            if i >= num_samples:
                break
    if all(l == 0 for l in len_motifs):
        try:
            motif = next(motif_col)
        except StopIteration:
            break
```

Sequence of events:

1. **Start with a motif key** (e.g., `'6mer'`).
2. **Loop over bins in order**:
   - `group` = all rows in that bin.
   - `uniques` = motif values **not already used** in previously selected rows.
   - If any unique values exist, **sample 1 row** uniformly at random from those (with a **fixed seed** `seeds[i]`).
   - Append the 1-row pick to `samples` and increment `i`.
3. **After sweeping all bins:**
   - If **no bin had new unique motifs** (`all(l == 0 …)`), **rotate** to the next motif key (`'8mer'`, then `'10mer'`, then `'12mer'`).
   - If we've exhausted all keys (iterator raises `StopIteration`), **terminate early** to avoid infinite loops.

Why this design:

- **Fairness across bins:** visiting bins round-robin prevents bias toward dense bins.
- **Uniqueness:** the `~group[motif].isin(used)` filter prevents duplicate motifs leaking in.
- **Determinism:** `random_state=seeds[i]` makes each pick **replayable**.
- **Fallbacks:** rotating motif keys salvages progress when the current key is saturated.

7) Packaging the result

```
if samples:
    result = pd.concat(samples, ignore_index=True)
    result.drop(columns='bin', inplace=True)
    return result
else:
    return pd.DataFrame(columns=df.columns.drop('bin'))
```

What: Concatenates all per-pick DataFrames into a single result; removes the helper `bin` column.

Why: Returns a clean table ready to merge/concatenate across classes. If no sample is possible, return an **empty** frame with the right schema (good for downstream robustness).

8) Apply the sampler per class & combine

```
unbound_samples = sample_probes(unbound_probes, 33)
weak_samples    = sample_probes(weak_probes,    33)
strong_samples  = sample_probes(strong_probes,  33)

final_sample = pd.concat([unbound_samples, weak_samples, strong_samples],
ignore_index=True)
final_sample.to_csv('dataset_old.csv', index=False)
```

What:

- Draws **33** items from each class using the same rules, then concatenates to a **balanced** 99-row dataset and writes it to disk.

Why:

- Balancing removes class bias in training/evaluation.
- Saving the file captures the exact selected subset for reproducibility and sharing.

Design choices & how to tune them

- **Bin width (0.1):** Increase for sparser data (e.g., 0.2) or decrease to capture finer gradients.
- **Motif key order:** The current order `6mer→8mer→10mer→12mer` prioritizes stricter uniqueness early. Swap the order (e.g., start with `8mer`) to focus on full-site diversity.
- **Seed schedule:** `seeds[i]` makes the *i-th* pick deterministic. Use a different base seed list to regenerate a new but reproducible sample.
- **Per-class counts (33):** Raise or lower to control dataset size; beware of exhausting unique motifs.

Edge cases & failure modes

- **Too few unique motifs:** The loop will rotate motif keys; if all keys saturate, it exits early with fewer than `num_samples` rows. Handle this by reducing `num_samples` or relaxing uniqueness (e.g., drop the shortest k-mer).
- **Empty bins:** The sweep simply skips bins with no candidates; stratification still works across the remaining bins.
- **Floating point bin edges:** Using `round(..., 1)` aligns edges but small numeric jitter can still place borderline values in adjacent bins; acceptable for stratification but keep consistent rounding.

Validation checklist

- Verify **no duplicate motif** per chosen key(s) in the output.
- Check **per-bin coverage** counts to ensure stratification behaved as intended.
- Confirm the **class balance** (equal rows per class).
- Re-run twice: outputs should be identical (deterministic) if inputs unchanged.

Complexity notes

- Each sweep touches all bins and filters uniques against previously selected motifs. With `k` picks and `B` bins, cost is roughly ~O(k·B) plus set-membership overhead. For typical dataset sizes, this is fast; for very large sets, pre-index motif→rows and track used motifs in a set for O(1) membership.

Minimal variants

- **Strict 8-mer uniqueness only:** remove rotation, enforce on `'8mer'` only.
- **Quota per bin:** instead of one pass round-robin, assign a target count per bin based on bin size to maintain proportional sampling with uniqueness.
- **Weighted sampling:** sample probability ~ inverse of bin density to over-sample rare regions while keeping uniqueness.

TL;DR. The sampler walks intensity bins in order, picks one unique-motif row at a time with a fixed seed, and if a key saturates, it rotates k-mer keys to keep making progress—producing a balanced, diverse, and reproducible subset.