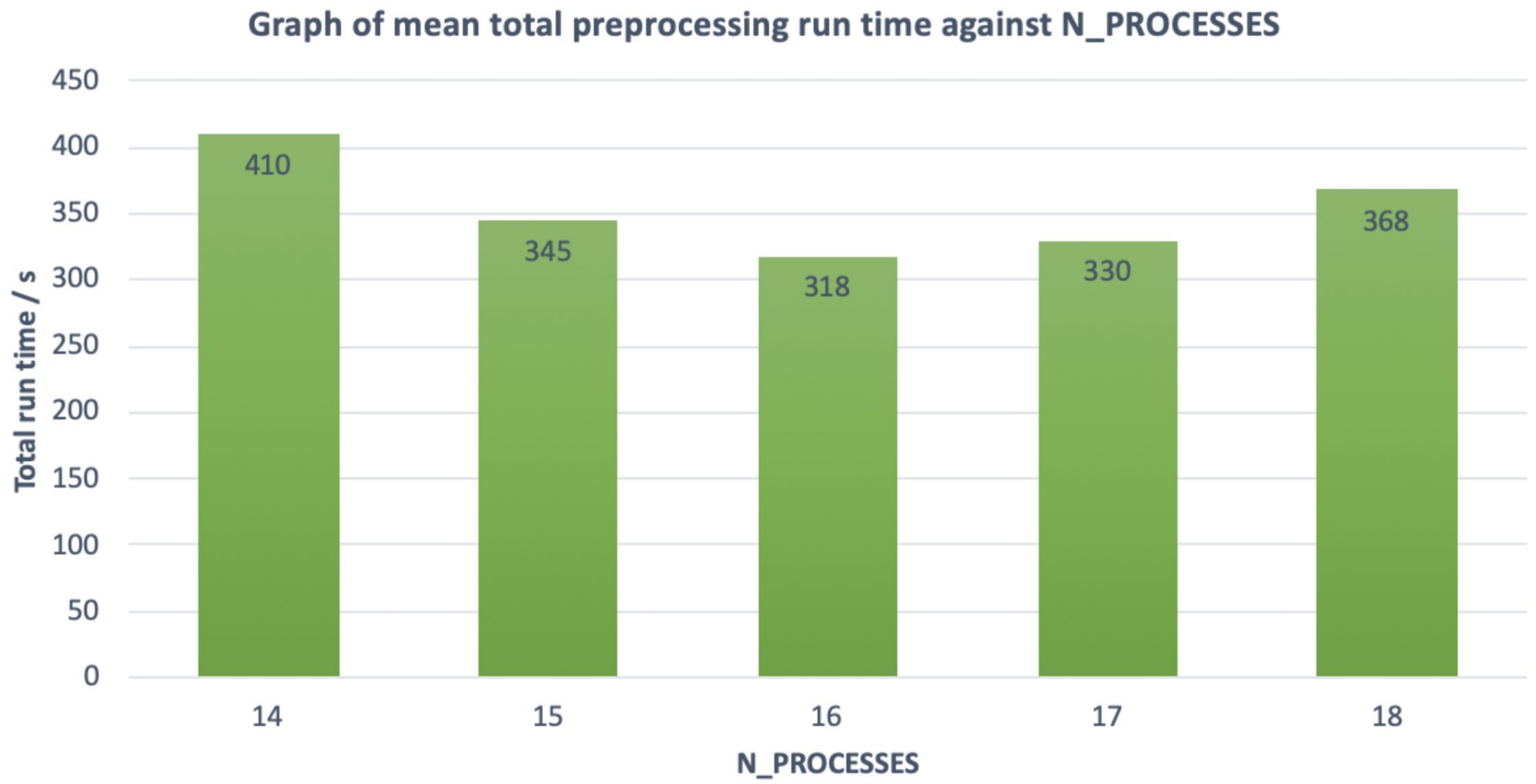
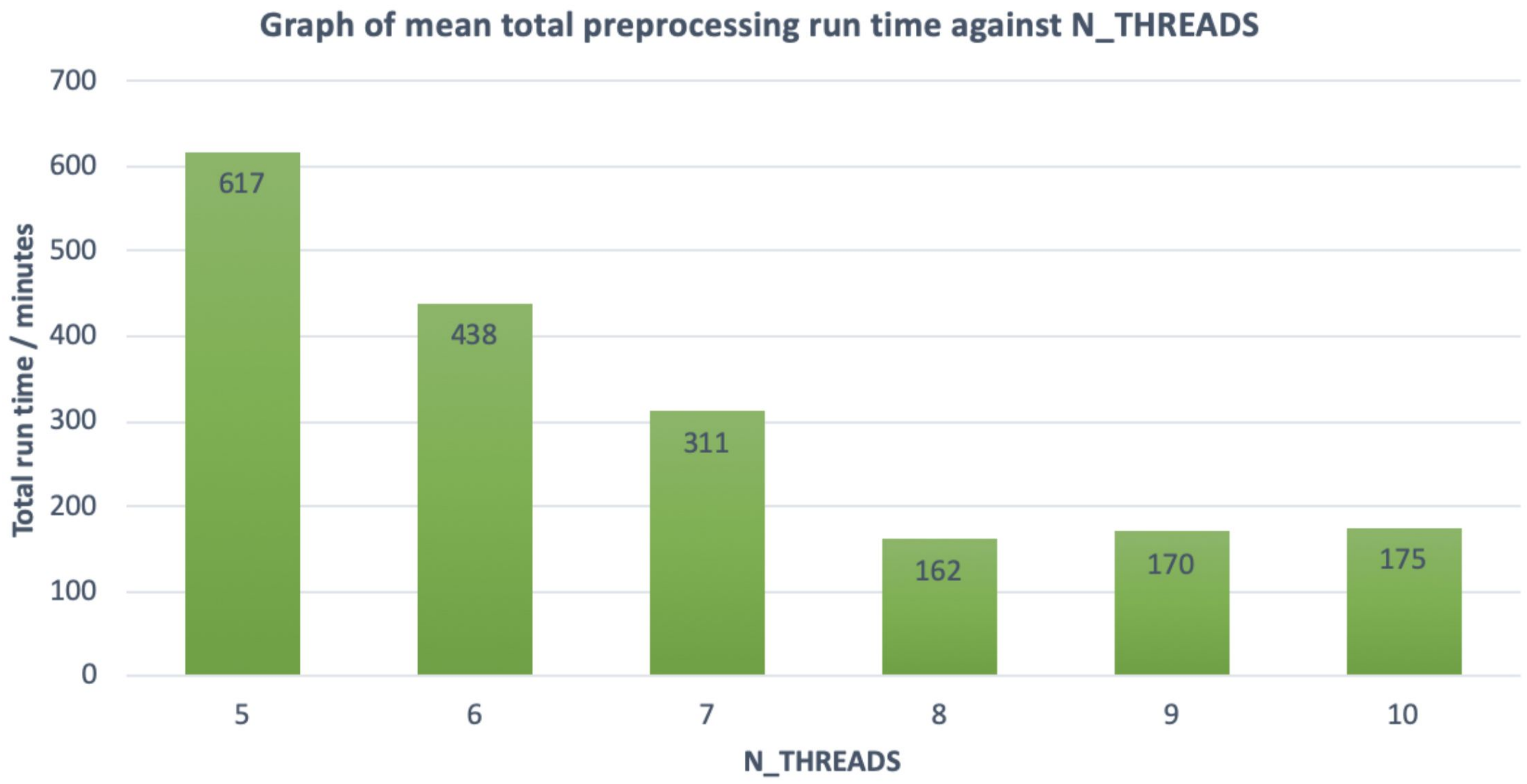
ncalls	tottime / s	percall / s	cumtime / s	percall / s	function
1	0.180	0.180	36751.569	36751.569	<module>
150	0.178	0.001	36748.625	244.991	attack_user
128	9.492	0.074	36049.427	281.636	user_data
57600	19.310	0.000	35753.019	0.621	extract_feats
57600	71.529	0.001	29391.213	0.510	tsfresh.extract_features
57600	27.916	0.000	28669.417	0.498	tsfresh._do_extraction
57600	18.031	0.000	21892.667	0.380	tsfresh.map_reduce
3144454	21767.459	0.007	21767.459	0.007	thread.lock.acquire
705716	7.104	0.000	21673.963	0.031	threading.wait
57600	2.091	0.000	6057.272	0.105	tsfresh.impute

ncalls	tottime / s	percall / s	cumtime / s	percall / s	function
1	0.035	0.035	752.915	752.915	<module>
1	0.028	0.028	752.519	752.519	main
1	3.806	3.806	531.462	531.462	remove_duplicate_roi_reports
168536	0.691	0.000	225.000	0.001	pandas.__getitem__
1	0.146	0.146	220.541	220.541	append_null_rois
168508	1.119	0.000	212.387	0.001	pandas._getitem_axis
167439	0.574	0.000	203.981	0.001	pandas.apply
167439	0.927	0.000	201.932	0.001	pandas.get_result
167439	2.674	0.000	200.140	0.001	pandas.apply_standard


Graph of mean total preprocessing run time against N_PROCESSES
Total run time / s
450
400
350
300
250
200
150
100
50
0
410
345
318
330
368
14
15
16
17
18
N_PROCESSES


Graph of mean total preprocessing run time against N_THREADS
Total run time / minutes
700
600
500
400
300
200
100
0
617
438
311
162
170
175
5
6
7
8
9
10
N_THREADS

ncalls	tottime / s	percall / s	cumtime / s	percall / s	filename:lineno(function)
1	0.000	0.000	18652.882	18652.882	<module>
1	0.000	0.000	18651.689	18651.689	run_attack
24	0.000	0.000	18650.340	777.098	threading.wait
548 186	50.340	34.033	18650.340	34.033	thread.lock.acquire
3	0.000	0.000	18650.333	6216.778	queue.join
1	0.125	0.125	1.335	1.335	get_ground_truth_matrix
1626/6	0.008	0.000	1.194	0.199	_find_and_load
1626/6	0.006	0.000	1.194	0.199	_find_and_load_unlocked
1315/2	0.005	0.000	1.193	0.596	_load_unlocked

Figure 7.3: A profile of our code for the DIFFERENT LOCATIONS attack run across all 150 sampled targets.

```
def geo_ind_df(epsilon: float, df: DataFrame, ID2LATLONG) -> DataFrame
    :
    planar_laplace_noise = PlanarLaplaceNoise(epsilon, ID2LATLONG)

    new_events = []
    for index, row in df.iterrows():
        new_point_id = planar_laplace_noise(10 * row.lat + row.long,
            SALT,
                row.epoch)
        new_events.append([row.target, new_point_id // 10,
            new_point_id % 10,
                row.epoch])

    return DataFrame(new_events, columns=COLUMNS)
```

Figure B.1: A sample from our code function which applies geo-indistinguishability to the SFC data.

```python
def suppress_low_counts(agg: ndarray, threshold: int) -> ndarray:
    for i in range(agg.shape[0]):
        for j in range(agg.shape[1]):
            if agg[i, j] < threshold:
                agg[i, j] = 0
    return agg
```

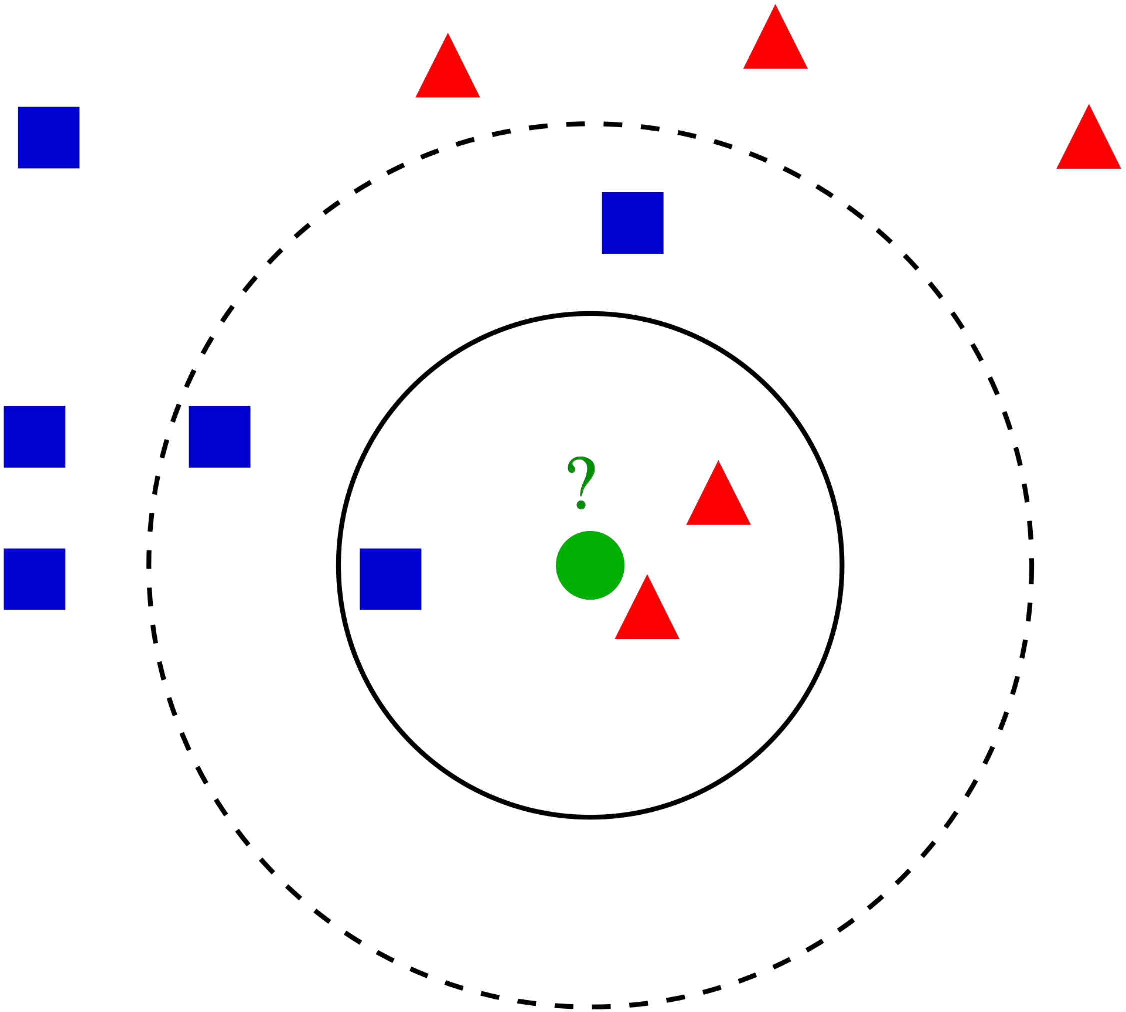
Figure B.2: A sample taken from the code which suppresses low counts in aggregates, and returns the aggregate which has had all low counts set to 0.

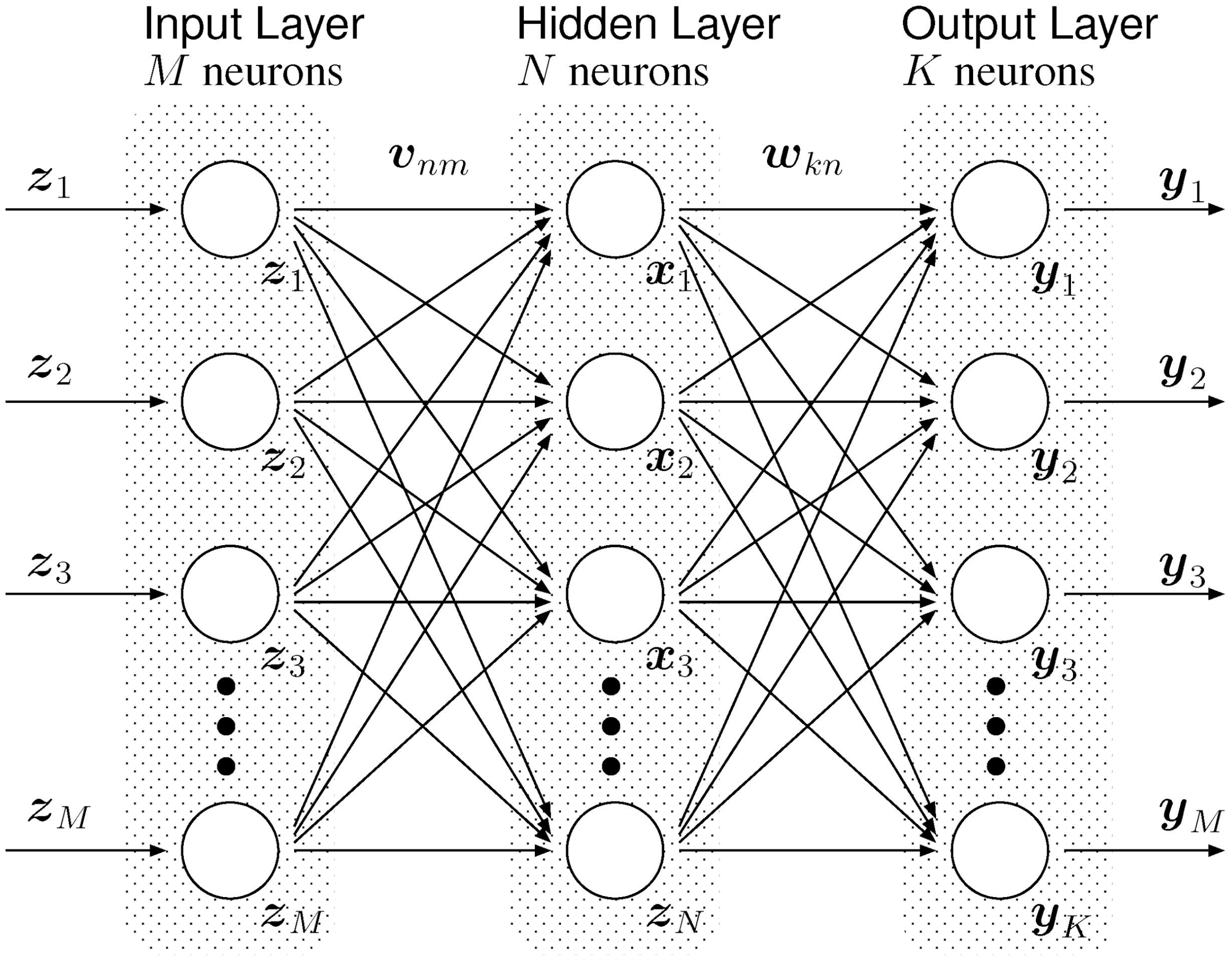
```python
def add_noise(
        agg: ndarray, noise_params: tuple, aggregation_size: int
    ) -> ndarray:
    assert aggregation_size > 0

    scaling = scale_factor(noise_params, aggregation_size)

    if noise_params[0] == "Laplacian":
        noise = random.laplace
    elif noise_params[0] == "Normal":
        noise = random.normal
    else:
        print("Error: Invalid noise function used. Exiting.")
        # Kill the whole program.
        sys.exit()

    perturbed_agg = zeros(agg.shape, agg.dtype)
    agg_x_len, agg_y_len = agg.shape
    for i in range(agg_x_len):
        for j in range(agg_y_len):
            perturbed_agg[i, j] = noise(agg[i, j], scaling)
    return perturbed_agg
```

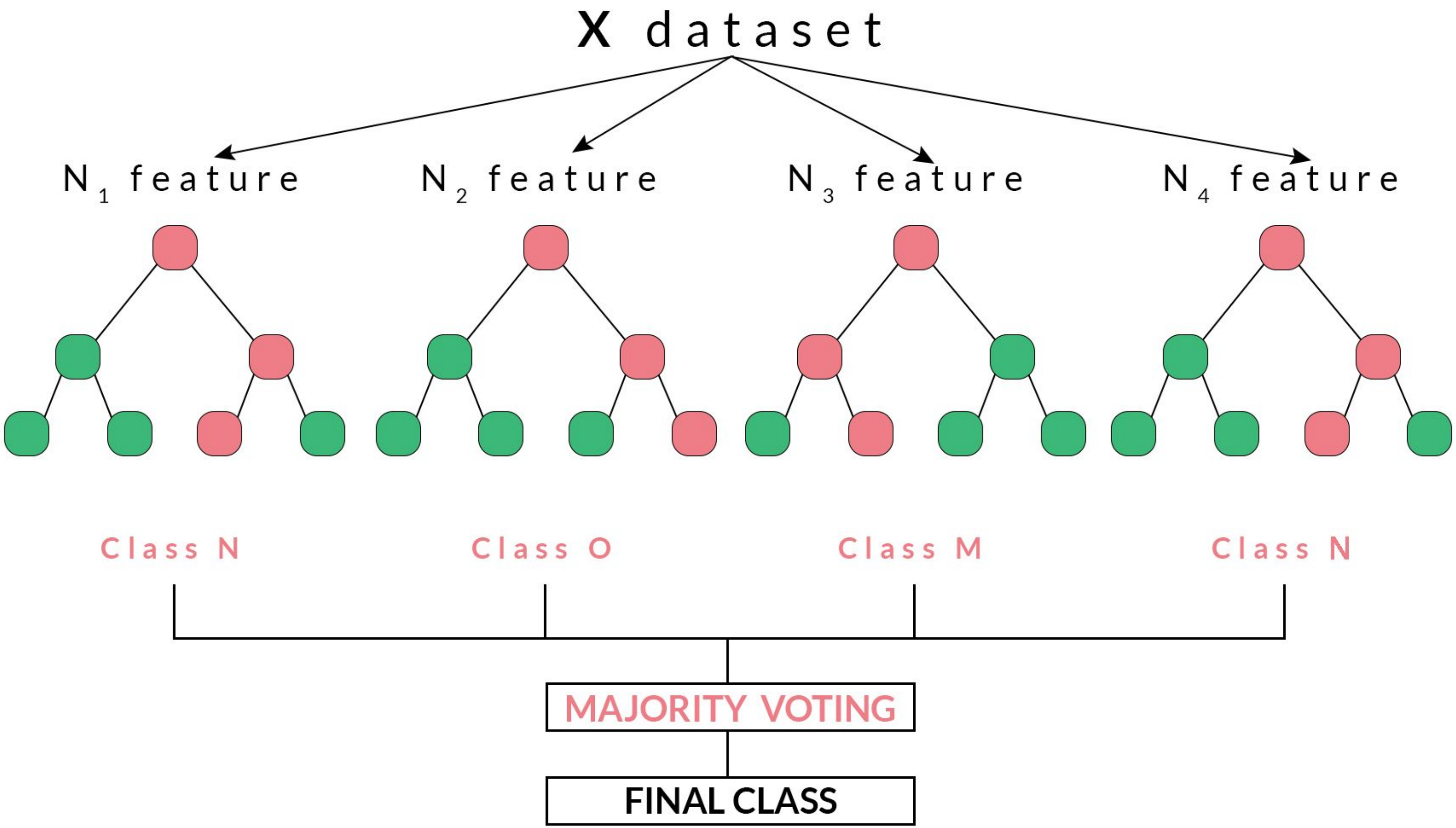
Figure B.3: A sample taken from the code which adds noise of some kind to an aggregate.

```python
def remove_duplicate_roi_reports(target_data):
    target = target_data['target'].iloc[0]
    rows = []
    for epoch in set(target_data.epoch.unique()):
        subset_df = target_data[target_data.epoch == epoch]
        if DATA_SET == "CDR":
            subset_df = subset_df.loc[:, ["point_id"]]
        if DATA_SET == "SFC":
            subset_df = subset_df.loc[:, ["lat", "long"]]

        # Get the single representative location.
        if DEFENSES["One ROI per epoch"] == "Mode":
            location = subset_df.apply(tuple, 1).mode()[0]
        if DEFENSES["One ROI per epoch"] == "Random":
            location = subset_df.apply(tuple, 1).sample(1).iloc[0]

        if DATA_SET == "CDR":
            rows.append([target, location[0], epoch])
        elif DATA_SET == "SFC":
            rows.append([target, location[0], location[1], epoch])


    return DataFrame(rows, columns=COLUMNS)
```

Figure B.4: The code used to implement single ROI defenses: WINNER TAKES ALL and random ROI reports.




Input Layer
M neurons
Hidden Layer
N neurons
Output Layer
K neurons
v_{nm}
w_{kn}
z_1
z_2
z_3
z_M
x_1
x_2
x_3
z_N
y_1
y_2
y_3
y_K
y_M


X dataset
N 1 feature
N 2 feature
N 3 feature
N 4 feature
Class N
Class O
Class M
Class N
MAJORITY VOTING
FINAL CLASS

How can we improve the attack?

1. Parametrization of the classifiers
2. Sampling high mobility users / targets
3. “Maybe…”: More training data

How else can we defend against the attack?

1. Restrict volume of training data (query denial)
2. ε differential privacy
3. Time generalization methods

How would we optimize the attack more?

1. Optimization of the TSFresh API function
2. More fine-grained jobs, threading
3. Better hardware
4. Use caches, not disk memory

How would you test your hypotheses?

1. Hypothesis test
2. Assume the AOC is $N(\mu, \sigma^2)$, and compute σ
3. Adapt our sampling in order to produce an experiment.

“¿How sure are you?”

- Unit tests on the components we added
- Type checking & integration tests with dummy data
- Repeat experiments with resampled targets

But:

- APIs untested
- Original code untested (although it’s already purportedly generalized)
- We do see variation in our results with sampling differences.

About the AOC

- Unbiased estimator of the *true* mean of the sample
- Discrepancies in small level comparison
- Important to consider stat significance

“Privacy gain” (Pyrgelis et al. 2017)

$$\mathrm{PL} = \begin{cases} \frac{\mathrm{AUC}-0.5}{0.5} & \text{if } \ \mathrm{AUC} > 0.5 \\ 0 & \text{otherwise} \end{cases}$$