Shae Herrod

Data Privacy and Security

Homework 2

1. October 2025
2. Laplace Mechanism
   1. discussion of the global sensitivity and corresponding noise amount

* Global Sensitivity: I use an average-over-S sensitivity Δ = (range) / m (where range = max\_age − min\_age and m = |{age>25}|). This is a dataset-independent sensitivity Δ = R / min\_m (where [min\_m](vscode-file://vscode-app/c:/Users/seamu/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html) is the minimum m across the original + three neighbor datasets) so the same GS is used for datasets, This is done so that the range of our noise sampling does not leak information about the dataset used.) Choosing a realistic static range of ages to bound our sampling range is incredibly important for maintaining utility as a very large range which includes many values not in the dataset will cause a greater amount of noise to be injected to the noisy response and thus make the response less accurate (though very private). The global range to be used is set in my program using a cli argument, for all of my tests I use a range of 100.
* if (g\_global\_range > 0.0) {
* int m\_orig=0, m\_o=0, m\_a=0, m\_y=0;
* double avg; int min\_age, max\_age;
* bool ok=true;
* ok &= get\_stats(input\_path, m\_orig, avg, min\_age, max\_age);
* ok &= get\_stats(data\_dir + "/adult\_minus\_oldest.data", m\_o, avg, min\_age, max\_age);
* ok &= get\_stats(data\_dir + "/adult\_minus\_age26.data", m\_a, avg, min\_age, max\_age);
* ok &= get\_stats(data\_dir + "/adult\_minus\_youngest.data", m\_y, avg, min\_age, max\_age);
* if (!ok) {
* cerr << "Warning: failed to compute stats for one or more datasets; falling back to per-file sensitivity." << endl;
* } else {
* int min\_m = max(1, min(min(m\_orig, m\_o), min(m\_a, m\_y)));
* fixed\_sensitivity = double(g\_global\_range) / double(min\_m);
* cout << "Using fixed sensitivity = global\_range / min\_m = " << fixed\_sensitivity << " (global\_range=" << g\_global\_range << ", min\_m=" << min\_m << ")\n";
* }
* }
* Noise sampling: the Laplace mechanism is used with scale b = Δ / ε; noise is drawn from Laplace(0,b) via the inverse-CDF

    double sensitivity;

    if (sensitivity\_override > 0.0) {

        sensitivity = sensitivity\_override;

    } else if (g\_global\_range > 0.0) {

        sensitivity = double(g\_global\_range) / double(max(1, m));

    } else {

        sensitivity = double(max\_age - min\_age) / double(max(1, m));

    }

    double scale = sensitivity / epsilon;

// Laplace noise generator using inverse CDF

double sample\_laplace(double scale, std::mt19937\_64 &rng) {

    std::uniform\_real\_distribution<double> unif(0.0, 1.0);

    double u = unif(rng);

    if (u == 0.5) return 0.0;

    if (u < 0.5) {

        return scale \* std::log(2.0 \* u);

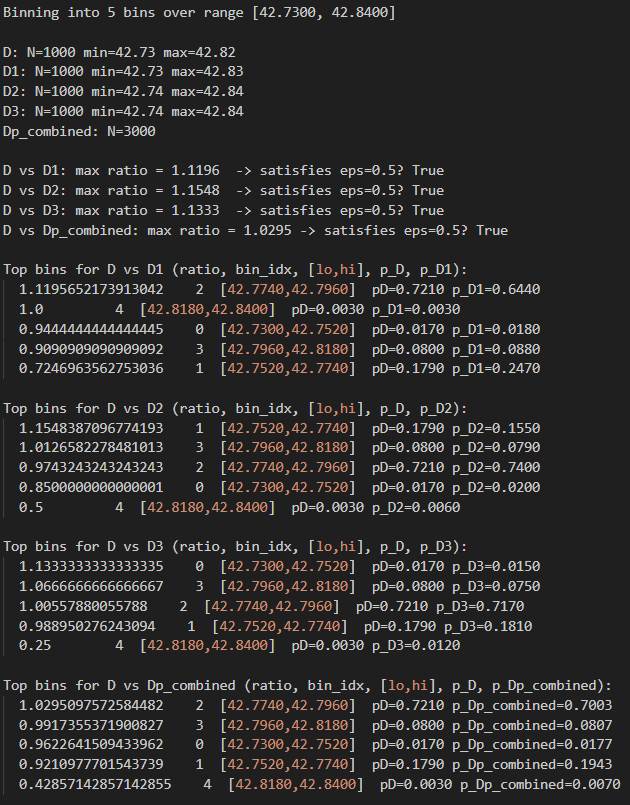
    } else {

        return -scale \* std::log(2.0 \* (1.0 - u));

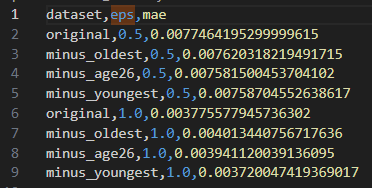
    }

}

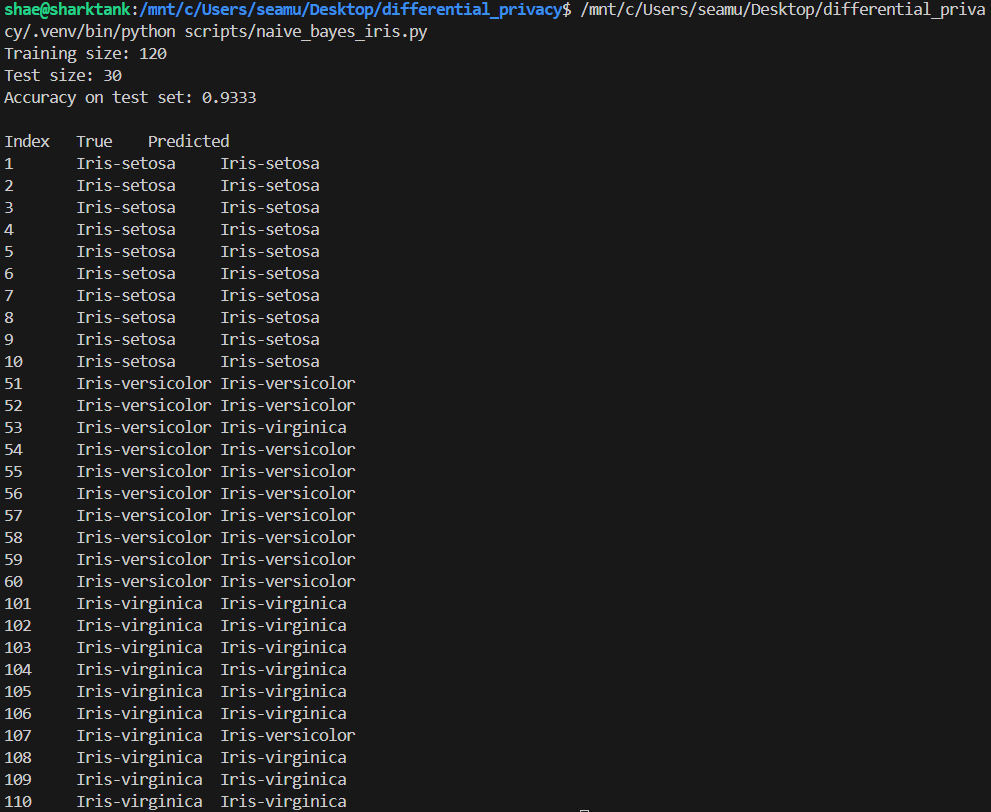
* 1. Validation of epsilon indistinguishability:

First I generate the neighboring datasets and run 1000 queries with each to get the data used for verification. The noisy outputs for each dataset is to two decimal places and saved to text files. I then switch to using python for writing the code used to validate. Then divided the output range into a set number of equal-width bins (in testing I found that n=5 was best so I include those results), counted how many results fell into each bin, and calculated the empirical probability for each bin by dividing the count by 1,000. To avoid division by zero, I applied smoothing by adding a small constant (alpha, such as 0.5) to each bin count. For each pair of datasets (original vs neighbor), I computed the ratio of probabilities for each bin and reported the maximum ratio observed. If all ratios are less than or equal to exp(0.5) ≈ 1.6487, the datasets are considered approximately 0.5-indistinguishable. If any ratio exceeds this threshold, possible causes include small sample size – 1000 samples is relatively few and without observing the results from many trials we could be getting an incomplete view of the true probability distributions for each bin, overly fine binning – with a larger number of bins there will be more ranges which contain zero entries from the dataset. 

1. The pipeline measures utility with Mean Absolute Error. For each noisy sample, x, produced by the DP mechanism it computes the absolute error |x − q(D)| where q(D) is the true query value (the true average of ages > 25 for that dataset). The mean absolute error is the arithmetic mean of those absolute errors across the trials (default 1000).
2. def true\_avg(path):
3. vals = []
4. with open(path) as f:
5. for l in f:
6. t = l.strip()
7. if not t: continue
8. tok = t.split(',')[0].strip()
9. try:
10. age = int(tok)
11. if age > 25:
12. vals.append(age)
13. except:
14. continue
15. return sum(vals) / len(vals)
16. # Load true averages
17. true\_avgs = {}
18. for key, fname in groups:
19. true\_avgs[key] = true\_avg(os.path.join(DATA\_DIR, fname))
20. # Load noisy values and compute errors
21. errors = {eps: {} for eps in noisy\_files}
22. mae = {eps: {} for eps in noisy\_files}
23. for eps, mapping in noisy\_files.items():
24. for key in mapping:
25. path = mapping[key]
26. vals = read\_noisy(path)
27. if len(vals) == 0:
28. errors[eps][key] = []
29. mae[eps][key] = None
30. continue
31. ta = true\_avgs[key]
32. errs = [abs(v - ta) for v in vals]
33. errors[eps][key] = errs
34. mae[eps][key] = statistics.mean(errs)



1. Differentially Private Classification:
   1. Screenshot of building and running, as well as testing results



* 1. Design and implement a differentially private algorithm

Algorithm:

1. Split train / test
   * Fixed test indices: rows 1–10, 51–60, 101–110 (1-based). Train on the rest.
2. Determine feature ranges
   * Compute per-feature min/max on the training set and use these ranges to clamp feature values before aggregation.
3. Allocate privacy budget
   * split total ε into three parts: epsilon\_count = 0.2·ε, epsilon\_sum = 0.4·ε, epsilon\_sumsq = 0.4·ε. (this is sequential heuristic within each class)
4. Compute true per-class sufficient statistics (non-private on training data)
   * For each class c:
     + true\_count = number of training records with label c
     + for each feature j: true\_sum[j] = ∑ x\_j (clamped), true\_sumsq[j] = ∑ x\_j^2 (clamped)
5. Compute sensitivities
   * count sensitivity = 1
   * sum sensitivity (feature j) = feat\_max[j] − feat\_min[j]
   * sumsq sensitivity (feature j) = feat\_max[j]^2 − feat\_min[j]^2
6. Add Laplace noise (Laplace mechanism)
   * For each class c and each scalar statistic:
     + noisy\_count = true\_count + Laplace(sens\_count / epsilon\_count)
     + noisy\_sum[j] = true\_sum[j] + Laplace(sens\_sum[j] / epsilon\_sum)
     + noisy\_sumsq[j] = true\_sumsq[j] + Laplace(sens\_sumsq[j] / epsilon\_sumsq)
7. Compute DP estimates (means, variances, priors)
   * Per-feature DP mean\_j = noisy\_sum[j] / noisy\_count
   * Per-feature DP E[x^2] = noisy\_sumsq[j] / noisy\_count
   * Per-feature DP variance = E[x^2] − (mean\_j)^2; floor small/negative variance to tiny positive value (1e-6)
   * DP class prior = noisy\_count / sum\_noisy\_counts
8. Build DP model
   * Store per-class: noisy\_count, mean vector, variance vector, prior.
   * Return model and clamp ranges.

Composition Analysis:

The queries (counts, sums, sumsq) are computed for each class. Counts across classes are disjoint (each training record contributes to exactly one class count), so the counts benefit from parallel composition over classes and can be released with epsilon\_count each (no extra sequential cost across classes). The same holds for sums and sumsq when computed only on disjoint subsets (per-class). However, since we release counts, sums, and sumsq for the same class, we must account sequential composition per class: for each class the budget used is epsilon\_count + epsilon\_sum + epsilon\_sumsq. In total the algorithm satisfies epsilon-differential privacy by construction (partition epsilon accordingly).

Privacy Budget Allocation:

The DP trainer privatizes per-class sufficient statistics (for each class it releases: count, per-feature sum, per-feature sum-of-squares). If each record in the training set belongs to exactly one class and the neighbor relation does not move a record from one class to another (i.e., neighbors differ only by adding/removing a record, not by changing its label), then the records used to compute class A’s statistics are disjoint from those used to compute class B’s statistics. To satisfy parallel composition, when you apply εi‑DP mechanisms to disjoint subsets of the data, the overall release satisfies max\_i εi (not the sum). So privatizing each class with budget ε (the same budget per class) yields overall ε (not |classes|·ε).

* 1. Set = 0.5, 1, 2, 4, 8, 16, and then calculate precision and recall of prediction results (screenshot of results on next page, after discussion)

The results show a clear trend of epsilon increases producing more accurate naïve bayes classifiers. Intuitively, this makes sense as a smaller epsilon corresponds to higher privacy requirements and more noise injected, so as epsilon increases (privacy relaxes), less noise is introduced to the system and thus the classifier has more accurate data to work on.

