Differential Privacy Temporal Map Challenge: Sprint 3 (Final Submission Writeup)

Final Submission – Jim King

Background

This is my final submission for the Differential Privacy Temporal Map Challenge, Sprint 3. I use the basic approach of using histograms of combined related features to reduce sensitivity and use a proximity dictionary during post-processing to further improve accuracy. The proximity dictionary is a trip seconds estimate between each of the Community Areas.

Main Idea:

The main idea is to combine similar features in the pre-processing phase, create privatized histograms of the features, then during the post-processing phase create the simulated data. The individual taxis are created by simply counting the number of distinct taxi_ids, adding noise and then iterating through the privatized count. The number of trips per taxi_id is calculated by counting the distinct taxi_ids with k number of trips (k = 1-200) and adding noise to each bin.

A total of 5 queries are used:

1) Count of distinct taxi_ids
2) Count of distinct taxi_ids with k number of trips
3) Histogram of the **proximity-shift-pca-dca** feature by a sub-sample of taxi_id
4) Histogram of the **company-payment_type** feature by a sub-sample of taxi_id
5) Histogram of **fare_codes** feature by a sub-sample of taxi_id

A proximity dictionary is created containing a trip seconds estimate for each pca-dca combination. The proximity dictionary is used in the post-processing process to align the privatized data.

Pre-Processing:

The data is pre-processed to combine similar features. The features to be combined are:

1) spd feature = shift, pickup_community_area(pca) and dropoff_community_area(dca)
2) cp feature = company and payment_type
3) fare_code feature = fare, tips, trip_seconds, and trip_miles

The fare code data (fare, tips, trip_seconds and trip_miles) is binned prior to combining into one feature and a proximity dictionary is created.

All pre-processing occurs in the pre-process function in utilities.py (lines 92-163). A proximity dictionary Is created for use in post-processing (lines 94-105).

Pseudocode for proximity dictionary:

For each pca-dca pair:
  Create dictionary entry {pca-dca: average trip_seconds}

Three new features are created by combining related features into one column.

1) shift-pca-dca (lines 108-115) – The columns shift, pca and dca are converted to strings, concatenated and converted back to an integer creating a single number/code representing a single combination of the features.
2) company-payment (lines 118-122) – The columns company and payment_type are converted to strings, concatenated and converted back to an integer creating a single number/code representing a single combination of the features.
3) fare_codes (lines 125-161) – The columns fare, tips, trip_seconds, trip_miles are converted to number/codes as follows. A dictionary of bins:labels is created based on a domain. This dictionary is then used to convert the column value into a bin code using the function convert_fare in utilities.py. These bin codes are then converted to strings, concatenated and converted back to an integer creating a single number/code representing a single combination of the features.

Pseudocode for combining features:

Concatenate (feature1) (feature2) (featureN)

These new columns are simply new representations of the data and do not represent queries on the ground truth.

Privitization

A total of 5 queries are used:

1) Count of distinct taxi_ids
2) Count of distinct taxi_ids with k number of trips
3) Histogram of the **proximity-shift-pca-dca** feature by taxi_id
4) Histogram of the **company-payment_type** feature by taxi_id
5) Histogram of **fare_codes** feature by taxi_id

The first two queries of drivers and trips are direct counts of taxi_id and are considered counting queries with sensitivity of 1.

1) **Count of distinct taxi_ids** - main.py (line 65-67) – The number of distinct drivers (taxi_ids) is calculated and privatized using the Laplace mechanism. (driver population count)
2) **Count of distinct taxi_ids with k (k=1-200) number of trips** – In utilities.py the following code creates the population and weights for trips:
    a. set_pop_weight_trips (lines 183-198) – Creates the population and weights for the trips.
    b. raw_weight_trips (lines 42-59) – Create the weights for 200 trip bins by counting the number of distinct taxis with k trips (k = 1-200).
    c. privatize_trips (lines 63-77) – privatizes the trip bins by adding Laplace noise.

For the remaining 3 queries, histograms are created. The histograms are essentially a population (bins) and weights. The creation of the population (bins) is simply a list of unique values of the feature converted into bins. For example, the **shift-pca-dca (spd)** feature is a list of all the unique **spd** values converted into bins.

The creation of the weights corresponding to each population (bins) is the key to ensuring the sensitivity of n is enforced. For each of the 3 histograms the following process occurs for creating the weights:

1) A dataframe is created with two columns – taxi_id and feature (e.g. **spd**)
2) All duplicate rows are dropped ensuring a one-one relationship between taxi_id and feature.
3) A new dataframe containing n samples (rows) from each taxi_id is selected
4) Weights are then created based on this new dataframe.

An example may help clarify this approach. Say I have a set of data with only 3 taxi drivers each with 100 trips for a total of 300 records and I want a sensitivity of 1 (n=1). Let's say there are 200 unique combinations of **shift-pca-dca**. The population is then 200 bins. The weights, however, only fill three bins as only one row for each taxi driver is selected. Since each taxi_id is only represented once in the histogram the sensitivity is 1. This approach will not perform well on a small set of data, but since the amount of data (the number of drivers) is very large the approach provides decent results.

The privatization for the histograms occur at the following points in the code:

1) main.py (lines 57-62) – For epsilons < 5.0 the sample size is set to 10 taxi_ids and for epsilons >= 5.0 the sample size is set to 30 taxi_ids. This represents the max records per individual for the histograms. The global sensitivity value is set to (3 histograms * sample size) + 2 population counts.
2) utilities.py (line 22 weight function) – The number of taxi_ids sampled is enforced on line 22.
3) utilities.py (line 28 weight function) - Each time weights for a histogram are calculated, the Laplace mechanism is used to add noise to bin value counts in the weight function.

Privacy Proof

Since a taxi driver is defined as the individual the *max_records_per_individual* is the number of taxi_ids used to calculate the weights when computing the sensitivity of taxi_id queries.

The sensitivity of function F is defined as: $\mathbf{max\ ||F(D_1) - F(D_2)\ ||_1}$

Where D_1 and D_2 are two data sets that differ by one individual and $||-||_1$ is the l_1 norm. For the histograms the l_1 norm is the total sum, or the absolute changes in bin counts, across all bins in the histogram, that occurs when we add or remove one person from the data set.

If 10 taxi_ids are used to calculate the weights in the histograms, then *max_records_individual* is 10. The number of histograms is 3 and there are two population queries.

The formula for calculating the sensitivity for building the simulated data is:

((max_records_individual x number of histograms) + population query) / epsilon

or

((10 x 3) + 2)/epsilon = 32/epsilon

Post-Processing:

Post processing consists of the following:

1) Iterating through the privatized number of taxis using 1000000 as the starting point (main.py line 80).
2) For each taxi, pulling a random trip count from the privatized trip count bins (main.py line 95).
3) For each trip, creating two halves of a simulated row using the privatized data (simulate row.py)
4) For the front half of the row containing pca and dca, the seconds estimate is added (simulate.py lines 35-27).
5) After iterating through all the taxis and trips, two dataframes from the two halves are created and combined as described below:

The dataframe concatenation process occurs in main.py lines 118-137:

1) A dataframe with the columns shift, company_id, pca, dca, payment_type and seconds estimate.
2) A dataframe with the columns fare, tips, trip_total, trip_seconds, trip_miles

Both dataframes are created in parallel from the privatized histograms and contain the same number of taxi_drivers and trips. Once completed, the first dataframe is sorted by the seconds estimate and the second dataframe is sorted by trip_seconds. The two dataframe are then concatenated together and the seconds_estimate column is deleted.

Proximity Dictionary

A dictionary of trip seconds estimates between each of the Community Areas is created in pre-processing for use in post-processing. A sample of the file is shown below.

{"0000": 879, "0001": 1305, "0002": 1231, "0003": 1462, "0004": 1289, "0005": 1582, "0006": 1499, "0007": 1504, "0008": 1307, "0009": 1051, "0010": 1200, "0011": 1484, "0012": 1065, "0013": 1202, …

The dictionary entry “0000”: 879 indicates the estimated seconds for a trip between area 00 and area 00 is 879. In other words, a taxi trip within area 00 is estimated to be 879 seconds.

The logic behind using this information is that trip_seconds is essentially a distance constant. A trip within area 08 will always be within a certain number of seconds (say 200 – 600 seconds) assuming there is little or no waiting time. Overall, trip_seconds should hold as a good proxy for distance within or between Community Areas.