Fuzzy SQL

Release 0.1.8

Samer Kababji @ EHIL

Oct 05, 2022

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 Motivation

Recently, aggregate queries are used as workloads to evaluate the utility of synthetic data [FLL+20]. The generated query is run on both real and synthetic datasets. The query results from both datasets are compared and the relative error is calculated to serve as a utility measure. However, the current studies assume pre-determined query formulation such as that proposed by Li et al where specific categorical variables are chosen for aggregation [LZL+18]. This deterministic approach leads to biased results since a synthesizer may learn the underlying distribution of some variables better than the others. To tackle this problem, we are inspired by the randomness introduced by SQL Fuzzing techniques also known as Fuzzers.

SQL Fuzzers are mainly used to test database management systems (DBMSs) for any bugs or vulnerabilities [noa]. Before executing an SQL query, a DBMS performs two levels of checks. First the SQL statement is checked for any syntactic error such as grammatical errors. Secondly, the query is checked semantically e.g. a call is made to a non-existent table. Once the DBMS performs the necessary checks, the SQL statement is executed according to the best execution plan [ZCH+20]. Fuzzers usually generate large amount of queries that do not pass the aforementioned checks. For instance, American Fuzzy Lop (AFL) [Won22], a widely used fuzzer, has only 30% out of it generated queries passing the syntax check while only 4% can pass the semantic check [ZCH+20]. While research attempts are made to focus on finding DBMS logic errors rather than semantic and syntactic errors such techniques generate both queries and the test datasets [GPR+20]. Other researchers [RS] propose an approach to generate queries that ensure fetching a randomly selected row and hence avoid syntactic and semantic error injection.

To ensure unbiased representation of synthetic data utility, we propose a Fuzzy SQL technique that have the following features: - Aggregate queries shall be randomly generated, i.e. grouping may be executed using any combination of the available categorical data. - Aggregation may take place across any of the available continuous variables. - A random condition may be imposed on the aggregate queries. In such case, the values used in the WHERE clause shall be randomly sampled from the real data and equally executed on both the real and synthetic data. - Datasets are available in tabular formal and may include categorical, continuous and date variables. - A proper metric shall be established to compare the results from the real and synthetic data.

It is important to pay a special attention to the metric to be used. For instance, Fan et al proposes1 to arrange the data with one of the categorical variables as a dependent variable. Then two classifiers are trained using training examples from the real and synthetic data respectively. Finally the testing examples from the real data are used to test both models using traditional metric such as F1. The F1 scores for both models are compared. Clearly, this approach will highly depend on the selection of the dependent variable especially if the dataset mostly incudes categorical variables.

1.2 Constructs

Fuzzy SQL is developed to enable the comparison between real and synthetic datasets that are generated by any ML technique. Fuzzy SQL will generate semantically and syntactically correct SELECT random queries that are simultaneously applied to both inputs of real and synthetic datasets. The tool returns all query results along with the corresponding query parameters such as the SQL statement. The query results may be further analyzed to measure distances between real and synthetic responses to each query.

The input datasets may be either tabular of longitudinal.

For tabular data, we define two basic types of SELECT queries, namely, 'filter' queries and 'aggregate' queries. For comparing real and synthetic datasets, usually 'aggregate' queries are used. This will allow the analyst to compare the resulting aggregate values from both datasets. Metrics, such as the Hellinger distance, can be applied to the aggregate query results in the same manner they are applied to the original datasets. On the other hand, an analyst may be interested to combine both filter and aggregate types of queries since filter queries may represent inclusion-exclusion criteria. Accordingly, we define a third type of queries and we call it 'filter-aggregate' query.

If the input data includes continuous variables, various 'aggregate functions' may be randomly applied in addition to counting the resulting number of records. These functions are limited to AVG, SUM, MIN and MAX. If the input doe snot include any continuous variable, only the COUNT aggregate function is applied. The types of each variable are typically defined by the user and inputted along with the real and synthetic datasets.

1.3 Tabular Data Templates

Without loss of generality, and to simplify the mathematical constructs, we herein ignore the logical operation 'NOT' and the value comparison operations BETWEEN, LIKE and IN. We further consider that the same set of value comparison operations is applicable to all types of variables. In practice, a distinction in their applicability is made and all the aforementioned operations are considered. Define:

$\mathcal{T}^r$: Database table for real data.

$\mathcal{T}^s$: Database table for synthetic data

N: The number of records in both $\mathcal{T}^r$ and $\mathcal{T}^s$.

$\mathbb{A}^n = \{A^n_1, A^n_2, \cdots, A^n_{|\mathbb{A}^n|}\}$ is the set of *nominal* variables in both $\mathcal{T}^r$ and $\mathcal{T}^s$ where $|\mathbb{A}^n|$ indicates the number of these variables.

$\mathbb{A}^c = \{A^c_1, A^c_2, \cdots, A^c_{|\mathbb{A}^c|}\}$ is the set of *continuous* variables in both $\mathcal{T}^r$ and $\mathcal{T}^s$.

$\mathbb{A}^d = \{A^d_1, A^d_2, \cdots, A^d_{|\mathbb{A}^d|}\}$ is the set of *date* variables in both $\mathcal{T}^r$ and $\mathcal{T}^s$.

For any member A_j in the above sets, it may assume a *value* given in the real dataset $\mathcal{T}^r$ such that:

$V(A_j)$ is the the set of all values that A_j may take. The length of $V(A_j)$ is $|V(A_j)| = N$.

We further define:

$LO = \{AND, OR\}$ is the set of logical operations.

$CO = \{=, \neq, <, \leq, >, \geq\}$ is the set of value comparison operations.

$AG = \{SUM, AVG, MIN, MAX\}$ is the set of aggregate functions.

Random samples are drawn from the above sets to construct the three major queries defined below. The basic sampling functions can be defined as:

$f_s : S_m \to S_s$ where f_s is a sampling function that maps any set S_m into a single element set S_s. For instance, the set AG may be mapped by f_s into $\{AVG\}$

$f_m : S_{m1} \rightarrow S_{m2}$ where f_m is a sampling function that maps any set S_{m1} into a multiple element set S_{m2}. For instance, the set $\mathbb{A}^n$ may be mapped by f_m into $\{A_1^n, A_{|\mathbb{A}^n|}^n\}$

1.3.1 Aggregate Queries

If $\mathbb{A}^c = \phi$, an aggregate query takes the form:

$$\begin{array}{rl} \text{SELECT} & f_m(\mathbb{A}^n), \text{COUNT(*)} \\ \text{FROM} & \mathcal{T}^r \\ \text{GROUP BY} & f_m(\mathbb{A}^n) \end{array}$$

However, if $\mathbb{A}^c \neq \phi$, an aggregate query takes the form:

$$\begin{array}{rl} \text{SELECT} & f_m(\mathbb{A}^n), f_s(AG)(f_s(\mathbb{A}^c)), \text{COUNT(*)} \\ \text{FROM} & \mathcal{T}^r \\ \text{GROUP BY} & f_m(\mathbb{A}^n) \end{array}$$

Similar queries are constructed for $\mathcal{T}^f$.

1.3.2 Filter Queries

If $\mathbb{A}^c = \phi$, a filter query takes the form:

$$\begin{array}{rl} \text{SELECT} & * \\ \text{FROM} & \mathcal{T}^r \\ \text{WHERE} & [f_s(\mathbb{A}^n \cup \mathbb{A}^c \cup \mathbb{A}^d) \quad f_s(CO) \quad f_s(V(f_s(\mathbb{A}^n \cup \mathbb{A}^c \cup \mathbb{A}^d)))] \\ & [f_s(LO)] \\ & [(f_s(\mathbb{A}^n \cup \mathbb{A}^c \cup \mathbb{A}^d) \quad f_s(CO) \quad f_s(V(f_s(\mathbb{A}^n \cup \mathbb{A}^c \cup \mathbb{A}^d))) \\ & f_s(LO) \quad f_s(\mathbb{A}^n \cup \mathbb{A}^c \cup \mathbb{A}^d) \quad f_s(CO) \quad f_s(V(f_s(\mathbb{A}^n \cup \mathbb{A}^c \cup \mathbb{A}^d))))] \\ & \ldots \end{array}$$

The WHERE clause comprises three basic expressions denoted by [ ]. The set length of the randomly selected query variables has an impact on these expressions. For instance, if $|f_m(\mathbb{A}^n \cup \mathbb{A}^c \cup \mathbb{A}^d)| = 2$, the first and second expressions are dropped and the SELECT statement will reduce to:

$$\begin{array}{rl} \text{SELECT} & * \\ \text{FROM} & \mathcal{T}^r \\ \text{WHERE} & [(f_s(\mathbb{A}^n \cup \mathbb{A}^c \cup \mathbb{A}^d) \quad f_s(CO) \quad f_s(V(f_s(\mathbb{A}^n \cup \mathbb{A}^c \cup \mathbb{A}^d))) \\ & f_s(LO) \quad f_s(\mathbb{A}^n \cup \mathbb{A}^c \cup \mathbb{A}^d) \quad f_s(CO) \quad f_s(V(f_s(\mathbb{A}^n \cup \mathbb{A}^c \cup \mathbb{A}^d))))] \end{array}$$

If $\mathbb{A}^c \neq \phi$, a filter query takes the form:

$$\begin{array}{rl} \text{SELECT} & f_s(AG)(f_s(\mathbb{A}^c)), \text{COUNT(*)} \\ \text{FROM} & \mathcal{T}^r \\ \text{WHERE} & [f_s(\mathbb{A}^n \cup \mathbb{A}^c \cup \mathbb{A}^d) \quad f_s(CO) \quad f_s(V(f_s(\mathbb{A}^n \cup \mathbb{A}^c \cup \mathbb{A}^d)))] \\ & [f_s(LO)] \\ & [(f_s(\mathbb{A}^n \cup \mathbb{A}^c \cup \mathbb{A}^d) \quad f_s(CO) \quad f_s(V(f_s(\mathbb{A}^n \cup \mathbb{A}^c \cup \mathbb{A}^d))) \\ & f_s(LO) \quad f_s(\mathbb{A}^n \cup \mathbb{A}^c \cup \mathbb{A}^d) \quad f_s(CO) \quad f_s(V(f_s(\mathbb{A}^n \cup \mathbb{A}^c \cup \mathbb{A}^d))))] \\ & \ldots \end{array}$$

1.3.3 Filter-Aggregate Queries

Filter-Aggregate queries are the most important for comparing real and synthetic datasets. The query is constructed by combining the above two forms. Hence, if $\mathbb{A}^c = \phi$, a filter-aggregate query takes the form:

SELECT $f_m(\mathbb{A}^n)$, COUNT(*)
FROM $\mathcal{T}^r$
WHERE $[f_s(\mathbb{A}^n \cup \mathbb{A}^c \cup \mathbb{A}^d) \quad f_s(CO) \quad f_s(V(f_s(\mathbb{A}^n \cup \mathbb{A}^c \cup \mathbb{A}^d)))]$
$[f_s(LO)]$
$[(f_s(\mathbb{A}^n \cup \mathbb{A}^c \cup \mathbb{A}^d) \quad f_s(CO) \quad f_s(V(f_s(\mathbb{A}^n \cup \mathbb{A}^c \cup \mathbb{A}^d)))$
$f_s(LO) \quad f_s(\mathbb{A}^n \cup \mathbb{A}^c \cup \mathbb{A}^d) \quad f_s(CO) \quad f_s(V(f_s(\mathbb{A}^n \cup \mathbb{A}^c \cup \mathbb{A}^d))))]$
$\ldots$
GROUP BY $f_m(\mathbb{A}^n)$

and if $\mathbb{A}^c \neq \phi$, a filter-aggregate query takes the form:

SELECT $f_m(\mathbb{A}^n), f_s(AG)(f_s(\mathbb{A}^c))$, COUNT(*)
FROM $\mathcal{T}^r$
WHERE $[f_s(\mathbb{A}^n \cup \mathbb{A}^c \cup \mathbb{A}^d) \quad f_s(CO) \quad f_s(V(f_s(\mathbb{A}^n \cup \mathbb{A}^c \cup \mathbb{A}^d)))]$
$[f_s(LO)]$
$[(f_s(\mathbb{A}^n \cup \mathbb{A}^c \cup \mathbb{A}^d) \quad f_s(CO) \quad f_s(V(f_s(\mathbb{A}^n \cup \mathbb{A}^c \cup \mathbb{A}^d)))$
$f_s(LO) \quad f_s(\mathbb{A}^n \cup \mathbb{A}^c \cup \mathbb{A}^d) \quad f_s(CO) \quad f_s(V(f_s(\mathbb{A}^n \cup \mathbb{A}^c \cup \mathbb{A}^d))))]$
$\ldots$
GROUP BY $f_m(\mathbb{A}^n)$

1.4 Longitudinal Data Templates

Write here definitions of tables and relations…

1.5 Metrics for Tabular Datasets

1.5.1 Hellinger Distance for Datasets

The Hellinger distance may be used to measure the quality of synthetic data. First we consider the calculation of the Hellinger distance between the real and the synthetic tabular datasets $\mathcal{T}^r$ and $\mathcal{T}^s$ respectively. Define:

$\mathbb{A} = \{A_1, \cdots, A_i, \cdots, A_{|\mathbb{A}|}\}$ is the set of *nominal* variables in both $\mathcal{T}^r$ and $\mathcal{T}^s$ where $|\mathbb{A}|$ indicates the number of these variables.

$o^j_{A_i}$ is the number of occurrences (i.e. counts) of the j^{th} class for the nominal variable A_i in $\mathcal{T}^r$. The discrete probability of the j^{th} class can be calculated as:

$$r^j_{A_i} = \frac{o^j_{A_i}}{\sum\limits_{\forall j} o^j_{A_i}}$$

For instance, consider the *nominal* variable $A_1 =$ "income" with two classes '<=50k' and '>50k'. Then the first class may have $o^1_{A_1} = 1200$ occurrences and the second may have $o^2_{A_1} = 2000$ occurrences with discrete probabilities of $r^1_{A_1} = 0.375$ and $r^2_{A_1} = 0.625$ respectively.

Similarly, for the synthetic data $\mathcal{T}^s$ we can calculate the discrete probabilities $s^j_{A_i}$

The Hellinger distance for the nominal variable A_i is calculated as:

$$\mathcal{H}^{A_i} = \frac{1}{\sqrt{2}} \left(\sum_{\forall j} \left(\sqrt{r^j_{A_i}} - \sqrt{s^j_{A_i}} \right)^2 \right)^{1/2}$$

The Hellinger distance between $\mathcal{T}^r$ and $\mathcal{T}^s$ can be calculated by taking the mean across all *nominal* variables:

$$\mathcal{H}^{\mathcal{T}} = \frac{1}{|\mathbb{A}|} \sum_{i=1}^{|\mathbb{A}|} \mathcal{H}^{A_i} \tag{1.1}$$

1.5.2 Hellinger Distance for Queries

In *aggregate* queries, grouping is done by randomly selected *nominal* variables. In this sense, measuring the Hellinger distance for the datasets as explained above is just a special case where grouping is done by a single nominal variable at a time. So, for $|\mathbb{A}|$ number of *nominal* variables in the original datasets, we may execute $|\mathbb{A}|$ number of queries with each query grouped by a single variable. Then by averaging the Hellinger distances of these queries, we reach the same results in (1.1)

If grouping is done by more than a single variable, it is as we are defining a new nominal variable A^q where A^q may be any combination of two or more dataset variables $A^i \quad \forall A^i \in \mathbb{A}$. The query will result in specific number of classes for A^q. Using the superscript j to indicate the j^{th} class of A^q, we calculate the Hellinger distance for the query by:

$$\mathcal{H}^{\mathcal{Q}} = \frac{1}{\sqrt{2}} \left(\sum_{\forall j} \left(\sqrt{r^j_{A^q}} - \sqrt{s^j_{A^q}} \right)^2 \right)^{1/2}$$

Both discrete probabilities r and s were defined earlier in *Hellinger Distance for Datasets*.

For instance, consider an aggregate query grouped by the two nominal variables A_1 = "income" and A_2 = "marital status" with each having two distinct classes. The query will result in the variable A^q having four distinct classes with a discrete probability $r^j_{A^q}$ for each resulting class j.

1.5.3 Euclidean Distance for Queries

Once the *aggregate* query is executed, the variable A^q, as defined in *Hellinger Distance for Queries*, will result in the classes: $1, 2..j..J$. If the data includes a continuous variable A^c, an aggregate function, say AGG, may be applied to that variable. For each class j, an aggregation value $[AGG(A^c)]_j$ of the continuous variable can be calculated. For instance, let A^q be a combination of two nominal variables A_1 = "income" and A_2 = "marital status". Let A^c = "age" be a continuous variable, then for each of the four distinct classes, we can calculate the AVG$($age$)$. Define:

v^r_j is the aggregate value (e.g. $[\text{AVG(age)}]_j$) corresponding to the j^{th} class of an arbitrary continuos variable A^c in $\mathcal{T}^r$.

v^s_j is the aggregate value corresponding to the j^{th} class of the same continuos variable A^c in $\mathcal{T}^s$

From the above components, we can find the difference components:

$d_j = v^r_j - v^s_j \quad \forall j$

We further find the mean and standard deviation across all the classes:

$\mu^d = \frac{1}{J} \sum_{j=1}^{J} d_j$

$\sigma^d = \sqrt{\frac{1}{J} \sum_{j=1}^{J} (d_j - \mu^d)^2}$

and we compute the standardized aggregate values:

$z_j = \frac{d_j - \mu^d}{\sigma^d}$

Finally, we compute the norm and normalize it to reflect the normalized Euclidean distance between the real and synthetic queries :

$\mathcal{E}^{\mathcal{Q}} = \frac{\|z_j\|}{J}$

Normalizing the distance by the number of resulting classes for the random query enables us to average the Euclidean distance across multiple queries since each of them may result in different number of classes.

CHAPTER

TWO

INSTALLATION

Install Fuzzy SQL using pip:

```
(.venv) $ pip install fuzzy-sql
```

Check out *Usage* for further information.

The package includes the necessary dependencies.

CHAPTER

THREE

USAGE

3.1 fuzz_tabular()

The core function for generating random queries is `fuzz_tabular()`. Here is a description of the function:

`fuzzy_sql.fuzzy_sql.`**`fuzz_tabular`**(*n_queries*, *query_type*, *real_file_path*, *metadata_file_path*, *syn_file_path='None'*, *run_folder='None'*, *printme=False*)

The function generates random queries for the input tabular datasets.

Parameters

- **n_queries** (*int*) – The number of random queries to be generated
- **query_type** (*str*) – The type of queries to be generated and can be 'single_agg', single_fltr', 'twin_agg', 'twin_fltr', or 'twin_aggfltr'
- **real_file_path** (*path*) – The full path to the real data csv file including the file extension.
- **metadata_file_path** (*path*) – The full path to the metadata json file including the file extension.
- **syn_file_path** (*path*) – The full path to the synthetic data csv file including the file extension or 'None' if random queries are desired for single dataset.
- **run_folder** (*str*) – The full path for the your output folder or 'None', which will save the output reports to the current folder.
- **printme** (*logical*) – Set it to True if an html report is desired. The report lists random records of all the generated queries.

Returns

A dictionary of all generated random queries.

Return type

dictionary

Here is an example how to generate 10 queries for a single dataset:

```
queries=fuzzy_sql.fuzz_tabular(10,,"single_fltr","path/to/file/X_real.csv", "path/to/
↪file/X_metadata.json")
```

Below is another example to generate 100 aggregate queries simultaneously applied to both real and synthetic input datatsets:

```
queries=fuzzy_sql.fuzz_tabular(100,"twin_agg","path/to/file/X_real.csv", "path/to/file/X_
↪metadata.json", "path/to/file/X_syn.csv")
```

Note: Windows users may need to add 'r' before the path string they pass to the function. This will force treating windows backslashes as literal raw character. For instance, pass: `r"C:\path\to\file\X_real.csv"`

3.2 make_table()

`fuzzy_sql.fuzzy_sql.`**`make_table`**(*table_name: str*, *df: DataFrame*, *db_conn: object*)

Imports the input dataframe into a database table.

Parameters

- **`table_name`** – The intended name of the table in the database.
- **`df`** – The input data
- **`db_conn`** – Database (sqlite3) connection object

3.3 load_csv()

`fuzzy_sql.fuzzy_sql.`**`load_csv`**(*file_path: Path*) → DataFrame

Reads the input csv file.

Parameters

`file_path` – The input file full path including the file name and csv extension.

Returns

The pandas dataframe in 'unicode-escape' encoding. Any "`" is deleted in the data.

3.4 TABULAR_QUERY()

`class` `fuzzy_sql.tabular_query.`**`TABULAR_QUERY`**(*db_conn*, *real_tbl_name: str*, *metadata: dict*)

This is a class used for generating random SELECT queries for a table residing in sqlite database.

Once called, various query parameters (listed below) are constructed typically in forms of dictionaries. The value entries of the dictionaries represent discrete probabilities for better model versatility. However, these probabilities can be modified only by accessing the constructor.

AGG_OPS: Aggregation functions

LOGIC_OPS: Logic Operations

NOT_OP_STATE: NOT operation state i.e. used or not used.

CAT_OPS: Value comparison operations for nominal variables.

CNT_OPS: Value comparison operations for continuous variables.

CAT_VAL_BAG: A bag of possible values for nominal variables.

CNT_VAL_BAG: A bag of possible values for continuous variables.

DT_VAL_BAG: A bag of possible values for date variables.

Parameters

- **`db_conn`** (*`connection object created using sqlite3.connect()`*) – A connection to the database that contains the table to be subjected to random queries.

- **real_tbl_name** (*String*) – The name of the table to be randomly queried.
- **metadata** (*Dictionary*) – A dictionary that includes table's variable names (i.e. column names) as keys and types of variables as values. THey types shall be restricted to: 'continuous', 'data' and 'nominal'. Any table shall have at least one nominal variable.

BIBLIOGRAPHY

[noa] SQLsmith: randomized SQL testing in CockroachDB. Section: testing. URL: https://www.cockroachlabs.com/blog/sqlsmith-randomized-sql-testing/ (visited on 2022-08-31).

[FLL+20] Ju Fan, Tongyu Liu, Guoliang Li, Junyou Chen, Yuwei Shen, and Xiaoyong Du. Relational data synthesis using generative adversarial networks: a design space exploration. *arXiv preprint arXiv:2008.12763*, 2020.

[GPR+20] Bogdan Ghit, Nicolas Poggi, Josh Rosen, Reynold Xin, and Peter Boncz. Sparkfuzz: searching correctness regressions in modern query engines. In *Proceedings of the Workshop on Testing Database Systems*, DBTest '20. New York, NY, USA, 2020. Association for Computing Machinery. URL: https://doi.org/10.1145/3395032.3395327, doi:10.1145/3395032.3395327.

[LZL+18] Kaiyu Li, Yong Zhang, Guoliang Li, Wenbo Tao, and Ying Yan. Bounded approximate query processing. *IEEE Transactions on Knowledge and Data Engineering*, 31(12):2262–2276, 2018. doi:10.1109/TKDE.2018.2877362.

[RS] Manuel Rigger and Zhendong Su. Testing database engines via pivoted query synthesis. URL: http://arxiv.org/abs/2001.04174 (visited on 2022-08-31), arXiv:2001.04174 [cs].

[Won22] Cerdic Wei Kit Wong. American fuzzy lop (afl) fuzzer. 2022.

[ZCH+20] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. Squirrel: testing database management systems with language validity and coverage feedback. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 955–970. 2020.

INDEX