Fuzzy SQL User Guide

Release 1.0.0-beta

Samer Kababji @ EHIL

Nov 10, 2022

CONTENTS

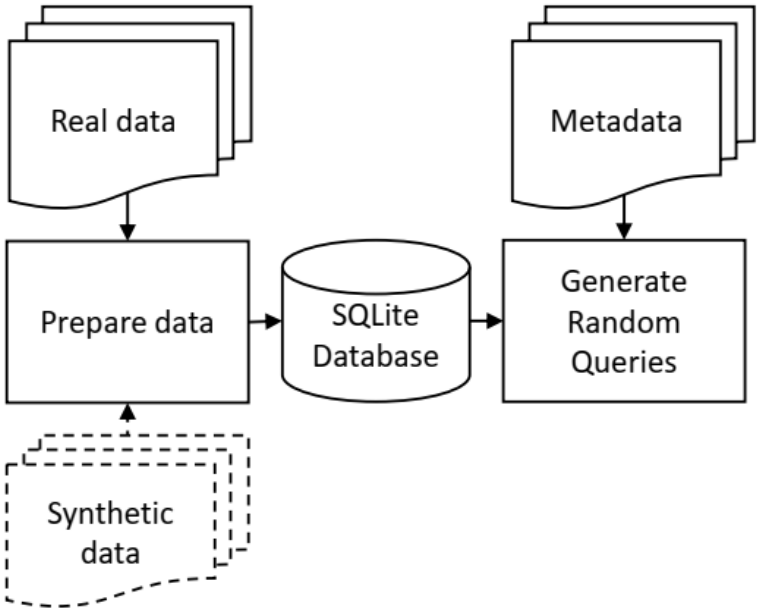
CHAPTER

ONE

INTRODUCTION

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.

A simplified flow is shown in the figure below. The dotted lines indicate that the synthetic data input is optional, although the package is mainly developed to compare the random query results for any real dataset and its synthetic counterpart. The input datasets may be either tabular or longitudinal.



We define two basic types of random SELECT queries, namely, 'filter' and 'aggregate' queries. We further combine these into a third type and we call it 'filter-aggregate' query. Metrics, such as the Hellinger distance, can be applied to the results of any 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 does not 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.

The package is versatile and allows the customization of various parameters including the type of logical and comparison operations and their associated discrete probabilities that are used to generate random queries. For instance,

assigning to the ‘AND’ logical operation a probability of 0.9 while assigning the ‘OR operation a probability of 0.1 will result in more AND’ed conditions than OR’ed in the WHERE clause.

CHAPTER

TWO

INSTALLATION

1. In your project directory, create your virtual environment and activate it. e.g. For Linux users, navigate to your project directory in the terminal and type

```
$ python3 -m venv .
$ source bin/activate
```

2. Make sure you update your pip. e.g. For Linux users, type:

```
$ pip install --upgrade pip
```

3. Install the Fuzzy SQL package using:

```
$ pip install git+ssh://git@github.com/skababji-ehil/fuzzy_sql.git@v1.0.0-beta
↪#egg=fuzzy_sql
```

The package includes all the necessary dependencies. Please note that the package is currently private and can be installed only by the personnel who have access to the repo.

Once installed, you may import any of the available functions. For further details, please check *Functions* and *Usage*

CHAPTER

THREE

FUNCTIONS

fuzzy_sql.fuzzy_sql.**prep_data_for_db**(*csv_table_path: Path*, *optional_table_name='None'*, *is_child=False*, *metadata_dir='None'*, *nrows=None*) → tuple

Reads the input csv file and prepares it for importation into sqlite db for fuzzy-sql analysis. By default, the file name (without extension) will be used as a table name in the database. All values are imported as strings. Any "" found in the values (e.g. '1') is deleted. Any variable (columns) that include dots in their names will be replaced by underscores.

Parameters

- **csv_table_path** – The input file full path including the file name and csv extension.
- **optional_table_name** – This is an optional name of the table when imported into the database. The default 'None' will use the csv file name (without extension) as the table's name.
- **is_child** – A boolean to indicate whether the input table is child or not. This will only impact the generated metadata template. Enter 'False' if the input table is tabular or not a child.
- **metadata_dir** – The directory where the metadata file shall be saved. No metadata file is saved if the default value of 'None' is used.
- **n_rows** – The number of rows to be read from the input csv file. The default of None will read all the rows in the csv file.

Returns

The pandas dataframe in 'unicode-escape' encoding. The corresponding metadata dictionary. The dictionary is saved to the chosen path as provided in metadata_dir.

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

Imports the input dataframe into a database table. All dots in the variable names will be replaced by underscores.

Parameters

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

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

Imports the input dataframe into an sqlite database table. The data will NOT be imported if it already exists in the database.

Parameters

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

fuzzy_sql.fuzzy_sql.**gen_queries**(*n_queries: int*, *db_conn: object*, *real_tbl_lst: list*, *metadata_lst: list*, *syn_tbl_lst: list*, *max_query_time=5*) → list

The function generates multiple twin random queries of aggregate-filter type.

Parameters

- **n_queries** – The required number of queries to be geenrated.
- **db_conn** – A connection to the sqlite database where all the input real and synthetic data reside.
- **real_tbl_lst** – A list of real tables to be used for generating the random queries. The list may include related tables.
- **metadata_list** – A lsit of dictionaries describing the varibales and relations for each input table. A single metadat dictionaries is used for each real table and its counterpart syntheitc table since both real and syntheitc tables shall have identical varibales and relations.
- **syn_tbl_lst** – A list of synthetic tables to be used for generating the random queries.
- **max_query_time** – The maximum time in seconds that is allowed to execute a randomly generated query expression before it skips it to the next random expression.

Returns

A list of dictionaries where each dictionary includes the query result for real data as a dataframe, the query result for synthetic data as a dataframe, a dictioanry describing the query details, a float represnting the twin query Hellinger distance and another represnting Euclidean distance, whenever applicable.

class fuzzy_sql.fuzzy_sql.**RndQry**(*db_conn: object*, *tbl_names_lst: list*, *metadata_lst: list*)

Generates a random query for tabular and longitudinal datasets.

Parameters

- **db_conn** (*object*) – The connection object of the sqlite database where the data exists.
- **tbl_names_lst** (*list of str*) – A list of input table names (strings) in the database to be randomly queried.

- **metadata_lst** (*list of dict*) – A list of dictionaries comprising the types of variables and relationships pertaining to each input table. Each dictionary shall conform to the metadata schema.

fix_seed

A boolean for setting the seed. Default is False and hence the query results will vary from one object to another.

oprtns: dict

A dictionary that defines the sets of various operations to be randomly sampled along with their desired discrete probabilities. The dictionary keys are defined below:

```
'AGG_OPS': The aggregate operations that can be used with any continuous␣
↪variable,if any, in the SELECT statement.
'LOGIC_OPS': The logical operations that can be used to combine conditions in␣
↪th WHERE clause.
'NOT STATE': A boolean with '1' indicating that a selected variable is to be␣
↪negated, i.e. preceded by NOT.
'CAT_OPS': The comparison operations that can be used with categorical␣
↪variables.
'CNT_OPS': The comparison operations that can be used with continuous variables.
'DT_OPS': The comparison operations that can be used with date variables.
'FILTER_TYPE': Whether to use WHERE or AND as a filter condition. This is␣
↪ignored in the case of tabular datasets since WHERE is the only choice.
'JOIN_TYPE': The type of JOIN in the SQL Lit database.  This is ignored in the␣
↪case of tabular datasets.
```

The default options are shown below. For any key, the sum of operation probabilities shall be 1. For instance, in the values below, the probability of sampling 'AVG' is higher than SUM, MIN and MAX, but they all sum up to 1. All these probabilities can be redefined by the user like all other attributes. However, the user needs to make sure that the assigned probabilities will always sum up to 1:

```
oprtns={
'AGG_OPS':{'AVG':0.25, 'SUM':0.25, 'MAX':0.25, 'MIN':0.25 },
'LOGIC_OPS':{'AND':0.9,'OR':0.1},
'NOT_STATE':{'0':0.9, '1':0.1},
'CAT_OPS':{'=':0.25, '<>':0.25, 'LIKE':0.15, 'IN':0.15, 'NOT LIKE':0.1, 'NOT IN
↪':0.1},
'CNT_OPS':{'=':0.2, '>':0.1, '<':0.1, '>=':0.1, '<=':0.1, '<>':0.1, 'BETWEEN':0.
↪2, 'NOT BETWEEN':0.1},
'DT_OPS':{'=':0.2, '>':0.1, '<':0.1, '>=':0, '<=':0, '<>':0.1, 'BETWEEN':0.2,
↪'IN':0.1, 'NOT BETWEEN':0.1, 'NOT IN':0.1},
'FILTER_TYPE':{'WHERE':0.5, 'AND':0.5},
'JOIN_TYPE': {'JOIN':0.5, 'LEFT JOIN':0.5}
}
```

max_in_terms: int

The maximum number of values to be used in the 'IN' operation. You can set that to np.inf if you do not want to enforce any upper bound.

no_groupby_vars: int

The fixed number of terms (vars) to be used in the GROUPBY clause. Set it to np.inf (default) if you need the number of terms to be randomly selected. If it is set to a larger number than the possible GROUPBY variables, then this number will be ignored.

no_where_vars: int

The fixed number of terms (vars) to be used in the WHERE clause. Set it to np.inf (default) if you need the number of terms to be randomly selected . If it is set to a larger number than the possible WHERE variables, then this number will be ignored.

no_join_tables: int

The fixed number of join terms (tables) to be used in the JOIN clause. It does not include the name of the master parent table (i.e. the table directly following 'FROM; in the SELECT statement). Set it to np.inf to randomly select the number of JOIN terms.

compile_agg_expr() → Tuple[str, list, str, list, tuple]

Generates random aggregate query expression.

make_single_agg_query(*single_expr: str*, *groupby_lst: list*, *from_tbl: str*, *join_tbl_lst: list*, *agg_fntn_terms: tuple*) → dict

Executes a single aggregate query expression and returns the result as a dataframe in a dictionary

make_twin_agg_query(*syn_tbl_name_lst: list*, *real_expr: str*, *real_groupby_lst: list*, *real_from_tbl: str*, *real_join_tbl_lst: list*, *agg_fntn_terms: tuple*) → dict

Executes a twin (both for real and synthetic datasets) aggregate query expression and returns the results as dataframes in a dictionary

compile_fltr_expr() → Tuple[str, str, list]

Generates random filter query expression.

make_single_fltr_query(*single_expr: str*, *from_tbl: str*, *join_tbl_lst: list*) → dict

Executes a single filter query expression and returns the result as dataframe in a dictionary

make_twin_fltr_query(*syn_tbl_name_lst: list*, *real_expr: str*, *real_from_tbl: str*, *real_join_tbl_lst: list*) → dict

Executes a twin filter query expression and returns the results as dataframes in a dictionary

compile_aggfltr_expr() → Tuple[str, list, str, list, tuple]

Generates a random aggregate-filter query expression.

make_single_aggfltr_query(*single_expr: str*, *groupby_lst: list*, *from_tbl: str*, *join_tbl_lst: list*, *agg_fntn_terms: tuple*) → dict

Executes a single aggregate-filter query expression and returns the result as a dataframe in a dictionary

make_twin_aggfltr_query(*syn_tbl_name_lst: list*, *real_expr: str*, *real_groupby_lst: list*, *real_from_tbl: str*, *real_join_tbl_lst: list*, *agg_fntn_terms: tuple*) → dict

Executes a twin aggregate-filter query expression and returns the results as dataframes in a dictionary

calc_dist_scores(*matched_rnd_query: dict*) → dict

Calculates Hellinger and Normalized Euclidean scores for the input random twin queries (i.e. real and synthetic) and updates the input dictionary with the calculated scores. The input queries shall be matched.

class fuzzy_sql.fuzzy_sql.**QryRprt**(*dataset_table_lst: list*, *random_queries: dict*)

Generates reports and plots for the input random queries.

Parameters

- **dataset_table_lst** (*list*) – List of table names that were used to generte the input queries.
- **random_queries** (*dict*) – A dictionary comprising multiple number of queries as datasframes with detailed desciption for each query.

query_to_html(*query_id: str*, *rnd_query: dict*) → str

print_html_mltpl(*output_file: Path*)

calc_stats() → Tuple[dict, dict]

plot_violin(*type: str*, *outputfile: str*)

CHAPTER

FOUR

USAGE

4.1 Schema

Whether tabular or longitudinal, data is passed to the main class of Fuzzy SQL and other functions as a list of individual tables. Each table shall be accompanied with its metadata dictionary which is typically read from a json file. The metadata dictionary includes the details about each variable in the corresponding table, any parent tables and their joining keys. For tabular data, the description of the parent tables can be simply omitted. Here are two hypothetical examples to clarify that.

- Tabular dataset:

  Assume that a tabular dataset is already imported into the Sqlite database as a table named **T** . The table **T** has two categorical variables *t1* and *t2* and one continuous variable *t3*. The metadata dictionary for **T**, namely **T_metadata** will look like the following:

```
T_metadata={
    "table_name":"T",
    "table_vars":[
        ["t1":"categorical"],
        ["t2":"categorical"],
        ["t3":"continuous"]
        ]
}
```

  The table name is passed to any applicable functions in Fuzzy SQL as a list. For instance, consider generating multiple random queries using the function *gen_queries* in *Functions* , you need to pass the table name as a list of single item, i.e. *real_tbl_lst=['T']*. Similarly, you pass its metadata as a list i.e. *metadata_lst=[T_metadata]*.

  Finally, any synthetic table names shall be passed in an identical way to that is used in passing real table names, i.e. in a form of list of table names. Please refer to *Code examples* for detailed code examples.

- Longitudinal dataset:

  Assume that a longitudinal dataset has a parent table **P** and a child table **C** that were already imported under the same names into the Sqlite database. **P** has the categorical variables *p1* and the continuous variable *p2*, while **C** has the categorical variables *c1*, *c2* and *c3*. The parent and child table are linked using the variables *p1* and *c1* respectively. The metadata dictionary corresponding to **P** will look like the following:

```
P_metadata={
    "table_name":"P",
```

(continues on next page)

(continued from previous page)

```
    "table_vars":[
        ["p1":"categorical"],
        ["p2":"continuous"]
        ]
}
```

while the dictionary corresponding to **C** will look like:

```
C_metadata={
    "table_name":"C",
    "table_vars":[
        ["c1":"categorical"],
        ["c2":"categorical"],
        ["c3":"categorical"]
        ],
    "parent_details":{
        "T":[["p1"],["c1"]]
    }
}
```

Notice that the above schema allows the addition of several parent tables as well as composite keys to be provided as list entries.

The table names are passed to various functions in Fuzzy SQL as a list of tables. For instance, consider generating multiple random queries using the function *gen_queries*, you can pass the dataset table names and dataset_dictionaries as lists of multiple items, i.e. *real_tbl_lst=['P','T']* and *metadata_lst=[P_metadata, C_metadata]* respectively. Please refer to *Code examples* for detailed code examples. The metadata schema can checked by accessing the class method: *RndQry._get_metdata_schema(self)*.

4.2 Data Types

To ensure the validity of the SQL select statement as interpreted by the database engine, Fuzzy SQL makes a distinction among three basic data types, namely: Categorical, Continuous and Date. Accordingly, if a dataset includes a variable with a different data type, it will be mapped to the proper type as per the table below:

Input Data Type	Output Data Type
'qualitative', 'categorical', 'nominal', 'discrete', 'ordinal', 'dichotomous', 'TEXT', 'INTEGER'	'categorical'
'quantitative', 'continuous', 'interval', 'ratio', 'REAL'	'continuous'
'date', 'time', 'datetime'	'date'

4.3 Code examples

Usage is best explained using real examples from various datasets. Please follow the steps below to install and run the examples:

1. Download and unzip the file from the link below:

   https://ehealthinformation-my.sharepoint.com/:u:/g/personal/skababji_ehealthinformation_ca/Ec6Paj0ypqNHm6_4cHn2qP4Br-ek5L6WGUGNar_tEf3oHQ?e=Nzrcxa

2. Navigate to the folder that contains the python files *main_sdgd.py, main_cal.py, main_cms.py and main_cms_tuned.py*. Each file is a standalone example and generates random queries corresponding to the following datasets:

   - sdgd: Tabular dataset
   - cal: Longitudinal dataset with single child
   - cms: Longitudinal dataset with multiple-child

3. In the directory above, create your virtual environment as explained in *Installation* . This is repeated here for convenience assuming a Linux system:

```
$ python3 -m venv .
$ source bin/activate
$ pip install --upgrade pip
$ pip install git+ssh://git@github.com/skababji-ehil/fuzzy_sql.git@v1.0.0-
↪beta#egg=fuzzy_sql
```

4. Run each of the four scripts using your activated environment above. The scripts are self-explanatory and include various useful comments.

INDEX