Structure of DashBot Implementation

SDC

October 26, 2021

Contents

1 Root

DashBot is implemented to run via a user interface as well as perform experiments. The folder implem contains the main part of the code, shared by both usages. experiment/ contains the code specific to experiments and interface/ the code specific to the interface.

In the root folder, you will also find the python script to execute:

- start-api.py for the interface
- start-experiment.py to run experiments.

2 Main (implem/ folder)

Interface (resp. experiment) runs with an instance of class Interface (resp. Experiment) that inherites from class DashBot.

Class DashBot is defined in implem/dashbot.py. It uses:

- modules, found in folder implem/modules/;
- binary files, found in folder implem/preprocessing_cache/, storing the result of the preprocessing step for each available dataset, if it has already been done once;
- csv files, found in implem/datasets/, if the binary file corresponding to the chosen dataset is not found ;

- implem/datasets/datasets.py links the name of the dataset (as asked by user) and the binary or csv file.

When initializing an instance of DashBot, the parameters are loaded into an instance of class Parameters (found in modules/parameters.py) from a yaml file found in experiments/ or interface/. The class Parameters allows to check if entered parameters are consistent and process them to be used into DashBot.

The main methods of DashBot are:

- preprocess_data(dataset_name);
- initialize_dashboard_generation();
- update_system();
- find_next_suggestion();

The first 2 methods are called once at the beginning, the last 2 after each user feedback.

2.1 Preprocess Data

When this method is called, an instance of class DataPreprocessor (found in modules/data_preprocessor.py) is first created. Then an instance of class AttributesRanking (found in modules/rankings.py) is created.

2.1.1 DataPreprocessor

During its initialization, 2 cases can be found:

1. the corresponding pickle file already exists:
   Then the result of preprocessing is loaded
2. the corresponding pickle file is not found:
   Then, the preprocessing is done and the pickle file is created.

The preprocessing result is:

- a list of instances of class Attribute (found in modules/attribute.py), for each attribute in the dataset. These instances have attributes:
  - directly extracted from the raw table (if it is numeric or not, etc.) ;
  - resulting from preprocessing computations (entropy, variance, class to rank them, if it has been discretized or not, etc.);
  - storing user feedback (if attribute is on the dashboard, if it has been reported as bad groupBy, the list of bad aggregation attributes that have been reported when it is on the groupBy dimension).
- the relation R^*, containing all data + new columns for discretized attributes, stored in a pd.DataFrame instance.

2.1.2 AttributesRanking

This instance will store rankings of attributes in terms of relevance.

- self.preprocessed:
  rankings for groupy and aggregation, as the direct result of preprocessing (no update depending on what happens during generation)
- self.general:
  rankings for groupy and aggregation, taking into account what happens during generation if it is a persistent information:
  1. bad groupby attributes reported by user are put in the 'bad' list, others in a 'good' list.
  2. attributes already on dashboard (if diversity is asked and not achieved) are put in 'less good' if previously in 'good', 'very bad' if previously in 'bad'.

- self.local:
  modification of self.general depending on the current state. These non-persitent modifications can be applied on e.g. :
  - bad aggregation attributes linked to the chosen groupby attribute(s).
    This depends on the current panel under construction. They are put in the 'bad' or 'very bad' list.
  - attributes that just have been reported by user explanation.
    This depends on the explanation given by the user during the last feedback, if applicable. They are completely removed from rankings and not just set as 'bad' or 'very bad' as they will be afterwards.

2.2 Initialize Dashbord Generation

This method initializes usefull instances for generation:

2.2.1 Diversity

- self.diversity['asked']: True/False, depending on if it is asked by the user (this is an option that can be changed by the user on the interface, this is a parameter that have to be put in the parameters file for experiments).
- self.diversity['achieved']: True/False, depending on the attributes that are already on the dashboard.

2.2.2 Dashboard

An instance of class Dash (found in modules/panel.py), which is an object representing the dashboard, i.e. containing several panels.

2.2.3 DashBot History

A list of the panels that have already been suggested. Each panel is stored as a list of integers corresponding to the vector representation of the panel.

2.2.4 Panel

An instance of class Panel (found in modules/panel.py), which is an object representing the current panel that is about to be / just have been shown to the user.
A panel can be either represented by:

1. its vector representation panel.vector
   a instance of pd.Series with a multi-index ('attribute', 'function').
2. objects containing attributes on groupby and aggregation dimensions:

   - panel.groupBy: list of instances of class Attribute
   - panel.aggregates: dict with instances of class Attribute as keys and list of function-strings as values.

For example, if dataset contains only attributes gender, age and BMI, the panel corresponding to the SQL query:

```
SELECT      gender, min(age), max(age)
FROM        dataset
GROUP BY    gender,
```

is either represented by:

1. the pd.Series panel.vector

*	gender					age					BMI				
count	gb	min	avg	max	sum	gb	min	avg	max	sum	gb	min	avg	max	sum
0	2	0	0	0	0	0	1	0	1	0	0	0	0	0	0

2. a combination of:
    - panel.groupBy = [Attribute(gender)]
    - panel.aggregates = { Attribute(age): ['min', 'max'] }

Depending on the algorithm used, modifications are made on one representation or the other. The untouched representation is then updated with methods panel.vector_to_attributes() if changes have been made on vector representation, panel.attributes_to_vector() otherwise.

Each panel shown to the user is identified by a panel_number in the form of a list of 2 integers. The first integer corresponds to the number of the panel on the dashboard that we are looking for (e.g. if 2 panels have already been validated, this number is 3) ; the second integer corresponds to the number of suggestions that have been shown to the user since the last validated panel (e.g. the sequence YES/NO/NO leads to a recommended panel with the second integer equal to 3).

2.2.5 User Feedback

user_feedback is a boolean variable representing the last user feedback: True means 'YES' ; False means 'NO' or 'NO + explanation'.

explanations_to_apply stores the explanation that have just been given by the user, if applicable ('NO + explanation'). It is empty if 'YES' or 'NO'.

2.2.6 Monitoring each iteration

Computations done between user feedback and next panel recommendation often use recursive functions that repeat an algorithm or cycle on attributes until panel requirements are fulfill. generation_counter, pairwise_panels_cycle and pairwise_panels_functions_to_remove keep track of this and ensures not to be trapped in infinite loops. It is mainly useful when performing experiments, where a large number of panels is recommended.

2.2.7 Setting parameters for MAB refinement

Depending on the parameters entered in the yaml file, internal variables are initialized:

- exploration_type in the case of ϵ-greedy algorithm;
- a list of explanation types (variable explanations) that will be used by softmax algorithm (see section 2.4.2).

2.3 Update System

When receiving user feedback, the system first updates internal variables before looking for the next recommendation.

- if 'YES', the last panel is added to the dashboard and diversity is checked (method validate_panel());
- if 'NO + explanation' with report of bad attributes, bad attributes lists are updated (method update_bad_lists()) ;
- if softmax have been used after the before last user feedback, softmax scores are updated according to the last user feedback (method update_explanations_scores()).

2.4 Find Next Suggestion

This is where panel_number is updated.

The method used to find the next suggestion is shared for all types of user feedback:

- if user_feedback is True (i.e. 'YES'), new panel generation module is launched ;
- if False and
    - explanations_to_apply is empty (i.e. 'NO'), MAB algorithm is performed ;
    - explanations_to_apply is not empty (i.e. 'NO + explanation'), the asked explanation is applied.

Each of these methods gives a panel and generation_counter is incremented (see section 2.2.6).

The panel is then confronted to all requirements:

- a panel has to be consistent with an SQL query (e.g. panels with no groupBy attribute are rejected) ;
- the same attribute can not be used on both dimensions ;
- the panel must not have already been recommended (method check_in_history())

Depending on the running algorithm, different strategies are used to fix or change a rejected panel (and incremente generation_counter).

When a good panel is found, it is shown to the user and added to dashbot_history (method show_to_U()).

2.4.1 New Panel Generation

generate_new_panel()'s goal is to find attributes for groupBy and aggregation. To do so, it resets the current panel and calls the method find_both_attributes() that loops on attributes in the order of the updated rankings of attributes.

1. Update ranking.local['groupBy']
2. Choose the best groupBy attribute
3. Update ranking.local['aggregation']
4. Choose the best aggregation attribute
5. Choose aggregation functions according to pairwise_panels_functions_to_remove
6. Incremente generation_counter
7. Check if the panel fulfill all requirements

If the panel is rejected, go to 4. with the second best aggregation attribute, etc. If all aggregation attribute have been tried, go to 2. with the second best groupBy attribute, etc.

This is where the distinction between 'good', 'less good', 'bad' and 'very bad' attributes is used (see section 2.1.2). It is referred in the code as gb_quality and agg_quality. DashBot first cycles on 'good' groupBy attributes + 'good' aggregation attributes. If all panels have been rejected, it cycles on 'good' groupBy attributes + 'less good' aggregation attributes, then 'less good'/'good', etc. The order of pairwise qualities is hard coded in the variable self.pairwise_qualities_2 of class AttributesRanking.

At this point, if all panels have been rejected, DashBot incrementes variable pairwise_panels_cycle (see section 2.2.6) and do the process again after changing how step 5. is done, i.e. the functions used to aggregate. Actually step 5. is performed in 2 steps:

1. choose_functions(agg_att) adds all functions to aggregate agg_att and, if possible, removes the bad functions associated to agg_att (reported by user explanations);
2. if possible, remove aggregation functions according to pairwise_panels_functions_to_remove and pairwise_panels_cycle.

2.4.2 MAB Algorithm

Method perform_MAB() is shared by the 2 MAB algorithms.

ϵ-greedy

1. pick a random number and compare it to parameter ϵ to discriminate between 'exploit' (exploit = True) and 'explore' (exploit = False).
2. apply changes
    - if 'exploit', choose number_of_bits = 1 bit and invert it (method refine_by_e-greedy(exploit)) ;
    - if 'explore' and parameter exploration_type

    = 'far-panel': choose several bits (several is number_of_bits, chosen according to parameter exploration_bounding) and invert them refine_by_e-greedy(exploit)) ;
    = 'new-panel': generate_new_panel() as if last panel were validated ;
    = 'hybrid': discriminate between 'far-panel' and 'new-panel' according to panel_number[1] and apply changes as in 'far-panel' or 'new-panel'.

3. fix panel (method fix_panel_for_MAB()) in the cases of no groupBy attribute, no aggregation attribute, same attribute on both dimension ;
4. incremente generation_counter ;
5. check if panel has already been suggested. If yes, repeat the whole process from the begining.

The method refine_by_e-greedy(exploit) chooses randomly the bit(s) to invert (choice with uniform probabilities), but forbids bad groupBy attributes (by putting their probabilities to be added as groupBy attributes to 0).

softmax This algorithm (method refine_by_softmax()) uses a notion of *explanation type*, defined by:

- the dimension on which the modification is made:
  among 'groupBy', 'aggregation' (i.e. all functions), 'min', 'avg', 'max', 'sum' ;
- if the modification is going to add or remove an attribute on this dimension.

Indeed, there are 12 types of explanations, declared as instances of the class Explanation (found in modules/explanation.py). In particular, these instances have an attribute score storing the success ratio of applying this type of explanation (success = directly followed by a feedback 'YES').
The algorithm runs as following:

1. find forbidden explanations (method find_forbidden_explanations()), e.g. "add max" if all attributes are already aggregated with 'max' ;
2. compute probabilities for each explanation type (method compute_probas()):
   - 0 if forbidden :
   - based on the success score, if not ;
3. choose an explanation type (method choose_explanation_type()) ;
4. randomly choose an attribute on which applying the chosen explanation type (method choose_attribute()) :
5. update variable explanations_to_apply accordingly and apply_explanation() as for feedback 'NO + explanation' (see section 2.4.3).

Step 3. is first done by randomly choosing an explanation type among the possible ones that have not been already applied (by softmax, but also by a feedback 'NO + explanation'), if possible. If not or once all explanation types are initialized, the choice is made based on their computed probabilities.

2.4.3 Explanation

1. apply asked changes to last panel (method clean_panel()) ;
2. check if same attribute is on both dimensions and if yes, remove one (method fix_panle_for_explanation()) If the applied explanation were "adding this attribute on dimension X", this method keeps dimension X and remove the other one ; if not, dimension is chosen randomly ;
3. check if the panel fulfills all the requirements
   and if not, try adding 1 attribute (method add_up_to_1_attribute()).
   This method works as the find_both_attributes() method (see section 2.4.1), with cycles on attributes to be added, but only on one dimension. To do so, agg_quality (resp. gb_quality) is None when adding attribute on the groupBy (resp. aggregation) dimension. The order of values for the tuple (gb_quality, agg_quality) is stored in the variable self.pairwise_qualities_1 of class AttributesRanking ;

4. if adding 1 attribute did not lead to a recommendable panel, add attributes on both dimensions (method find_both_attributes()), as for the new panel generation (see section 2.4.1).

Methods add_up_to_1_attribute() and find_both_attributes() take as arguments forbidden_groupBy and forbidden_aggregation, to prevent from adding again an attribute that have just been removed by the asked explanation.

3 Interface (interface/ folder)

3.1 Specificities of Interface Implementation

3.2 Front End

4 Experiment (experiment/ folder)

4.1 Data

Each run of an experiment generates a *run file* stored in an *experiment path.*

4.1.1 Run File

named <run_id>_<date>_<time>.csv
storing:

1. **header** : summary of parameters under which experiment is done (as in experiment_path) ;
2. **data** : row = 1 iteration, with:
   - iteration_number: number of iterations since dashboard generation started (start = 0) ;
   - algo: algorithm used to find suggested panel ("random", "new-panel", "MAB" or "explanation") ;
   - panel_number: number of target panels found - 1 ;
   - suggestion_number: number of panels suggested (start = 0) since a target panel have been found (for the first one: since the dashboard generation started) ;
   - distances: distances between suggested panel and target panels (ordered with their indexes), separated with '/' ;
   - iteration_time: computing time (ms) of iteration ;
   - total_time: computing time (ms) since dashboard generation start ;
   - found_target: if suggested panel is validated, index of target panel found ; if not, None ;
   - #generations: number of panels generated until one matches all requirements for being suggested.

4.1.2 Experiment Path

results/	
results files/	
<experiment type>/	type of experiment in 'MAB_parameters', 'paper_plots' or 'test'
<dataset name>/	name of dataset as in <dataset_name>.pkl
<target>/	names of target panels separated with '_'
<n attributes>/	number of attributes selected in relation table
<numeric ratio>/	ratio of numerical attributes
<attribute threshold>/	preprocess parameter
<diversity>/	True or False
<inclusion>/	True or False
<startegy>/	strategy used to suggest panels ('random', 'MAB', 'explanation', or combination of two of t
<algos ratio>/	hybrid strategy: <#algo1_#algo2>
	others: None
<explanation type>/	type of explanation modelisation:
	one: one in 'remove groupBy', 'remove aggregation', 'change functions' (priority order a
	all: all types can be given at once
	None: if <strategy> is not hybrid or explanation
<MAB algorithm>/	'e-greedy' or 'softmax', if <strategy> is 'MAB' or 'hybrid'
	None, otherwise
<epsilon>/	epsilon, if <strategy> is 'MAB' or 'hybrid'
	None, otherwise
<exploration type>/	'far-panel' or 'new-panel', if <MAB algorithm> is 'e-greedy'
	None, otherwise
<exploration bounding>/	exploration_bounding, if <exploration type> is 'far-panel'
	None, otherwise

4.2 Plots

Once all runs of an experiment are done, plots are generated as following:

4.2.1 Get Summary

script results/modules/get_summary.py:

1. checks if parameters in header are consistent with <experiment_path> ;
2. summarizes all runs of the experiment, stored as a row in file results/<experiment_type>/summary.csv:
   - parameters, as described in data path: dataset_name, target, n_attributes, numeric_ratio, attribute_threshold, diversity, inclusion, startegy, algos_ratio, explanation_type, MAB_algo, epsilon, exploration_type, exploration_bounding
   - results of experiment:
     - #iterations – mean of number of iterations to find target dashboard
     - #itertions_std – standard deviation of number of iterations to find target dashboard
     - time – mean of computing time to find target dashboard
     - time_std – standard deviation of computing time to find target dashboard

   <experiment_type> can be:

   - 'MAB_params', if experiment is run to look at the influence of MAB parameters ;
   - 'paper_plots', if experiment is run to plot 'fixed dashboard', 'fixed relation'
   - 'test', otherwise ;
3. plots some info and stores them in <experiment_path>:
   - time = f(#iterations)
   - distribution of #iterations

- distribution of time
- distribution of #generations_per_iteration
- distibution of time_per_iteration

4.2.2 Plot Summary

script results/modules/plot_summary.py

4.2.3 Plot F1 scores

script results/modules/plot_F1scores.py

Appendices

A Tunable Parameters