OREO: Detection of Cherry-picked Generalizations

Yin Lin
University of Michigan
irenelin@umich.edu

Brit Youngmann
CSAIL MIT
brity@mit.edu

Yuval Moskovitch
University of Michigan
yuvalm@umich.edu

H. V. Jagadish
University of Michigan
jag@umich.edu

Tova Milo
Tel Aviv University
milo@cs.tau.ac.il

ABSTRACT

Data analytics often make sense of large data sets by generalization: aggregating from the detailed data to a more general context. Given a dataset, misleading generalizations can sometimes be drawn from a cherry-picked level of aggregation to obscure substantial subgroups that oppose the generalization. Our goal is to detect and explain cherry-picked generalizations by refining the corresponding aggregate queries. We demonstrate OREO, a system to compute a support score of the given statement to quantify the quality of the generalization; that is, whether the aggregated result is an accurate reflection of the data. To better understand the resulting score, our system also identifies significant counterexamples and alternative statements that better represent the data at hand. We will demonstrate the utility of OREO for investigating generalizations, by interacting with the VLDB'22 participants who will use the OREO interface for statement validation and explanation.

PVLDB Reference Format:
Yin Lin, Brit Youngmann, Yuval Moskovitch, H. V. Jagadish, and Tova Milo . OREO: Detection of Cherry-picked Generalizations. PVLDB, 15(12): 3570 - 3573, 2022.
doi:10.14778/3554821.3554846

1 INTRODUCTION

Statements based on aggregate query results over datasets are commonly used by data scientists when analyzing large datasets. These statements, which we refer to as *generalizations*, allow analysts to represent, and convey, a high-level understanding of the data. For example, given a dataset of people with height and gender, we may arrive at a generalization that, on average, men are taller than women. Of course we know there are many exceptions – in pairwise individual comparisons, we will find many women taller than men. Nevertheless, the generalization may still be a "reasonable" conclusion from the data.

Misleading statements could be made intentionally, e.g., by cherry-picking generalization levels to obscure the information of subgroups opposing the statement. Examples can be found in many statements made by politicians. Poorly constructed generalizations, also called "hasty generalizations", may occur even in "objective" arenas like science and medicine, where they could affect the decision-making processes. For instance, doctors have overdiagnosed ADD and ADHD for years after making generalizations to age, sex, and the maturity of the children [4]. Undesirable generalizations might convey misleading information even if the statements are technically supported by the data.

To illustrate this problem we consider the following example.

Example 1.1. Nowadays, there are rising concerns over ageism in tech companies, which may lead to developers older than 35 feeling "over the hill" in the workplace [3]. We consider an extract from Stack Overflow, containing answers of users to the Stack Overflow developers survey, presented in Table 1. For simplicity, we consider only the Gender, Age (discretized), Role, and Salary attributes and construct an "Aged Over 35" attribute to indicate the age group of each respondent. The aggregate results indicate that respondents under 35 earn more on average compared with respondents over 35. However, this is not the case for significant sub-groups, e.g., designers and DB administrators. In view of these significant exceptions, the generalization on its own might not be an appropriate representation of the data.

To this end, we have previously designed a scoring framework [13] to quantify the quality of generalizations. Given a generalization derived from an aggregate query, the main idea is that we refine the query using conjunctions of predicates to explore the sub-groups for evidence of support. Our model assigns a score in the range of 0 to 1 to a given statement, where intuitively, the score of a statement reflects the degree to which the aggregate result represents the underlying data. To provide the user a better understanding of the resulting score, we could provide *counterexamples*, i.e., disclosing significant parts of the data opposing the statement. We could also refine the statement to obtain alternatives that better represent the data. In [13], we proposed algorithms to perform such tasks.

Given this toolbox of options developed in [13], we need to organize these options in a cohesive system that serves the needs of a user. It is this systems challenge that we undertake in this demonstration. We propose to demonstrate our solution, which we have implemented in a system called OREO (for "detectiOn of cheRry-pickEd generalizatiOns"). OREO provides an interactive UI that allows the users to examine the quality of statements based on aggregate queries. We provide an easy-to-use statement builder, which allows the users to explore their own generalized statements by specifying main statement components. To help the users better

Proceedings of the VLDB Endowment, Vol. 15, No. 12 ISSN 2150-8097.
doi:10.14778/3554821.3554846

Table 1: Example data set.

Res ID	Gender	Age	Aged Over 35	Role	Salary
1	Male	25 - 34 years old	No	Developer	94K
2	Male	25 - 34 years old	No	DB Admin	11K
3	Female	25 - 34 years old	No	Developer	86K
4	Female	18 - 24 years old	No	Designer	19K
5	Male	35 - 44 years old	Yes	Developer	13K
6	Male	45 - 54 years old	Yes	DB Admin	18K
7	Female	35 - 44 years old	Yes	Developer	14K
8	Female	35 - 44 years old	Yes	Designer	40K

understand the resulting score of a given statement, OREO generates counterexamples. Moreover, OREO can be used to generate possible high-scoring alternatives to low-score statements.

We note that while we provide the theoretical foundations and algorithms underlying the demonstrated system in [13], this demonstration presents OREO usability and its suitability for end-to-end deployment. In the rest of this demo description, we first present the technical background of our framework in Section 2, including the problem formulation and algorithms. We then provide the system details, including the system overview and the user interface in Section 3. In Section 4, we overview our demonstration plan.

Related work. While a rich vein of research [6, 10–12, 14, 15] has been devoted to computational fact-checking aiming to compare the claims against existing facts based on structured data, most of them formulate the correctness of the statement based on the dataset query results. Wu et al. [14, 15] explore data perturbation and provide an efficient framework to model claims as parameterized queries for robustness and accuracy evaluation. Jo et al. [10–12] aim at verifying aggregate data summaries from relational databases with a natural language interface. However, simply modeling the appropriateness of the generalization level via a single aggregate query or the perturbation of parameterized queries is not enough. As indicated in our motivating example, the query results might not be an accurate reflection of the real scenario. OREO provides a tool, which not only focuses on maliciously false claims but also on the cherry-picking scenario, where the aggregate results could be true but misleading. Related work of the cherry-picking statement detection includes [6], which focuses on evaluating cherry-picked trendlines, where "unreasonable" trends could be derived from falsely chosen endpoints.

2 TECHNICAL BACKGROUND

We provide a overview of our theoretical foundations of the development of OREO (see [13] for more details). We demonstrate the ideas using the example in Section 1.1.

2.1 Model

We start by presenting the notion of statements based on aggregate queries and then discuss their score, which reflects the degree to which the result of aggregation represents the underlying data.

Partition query. Let $\mathcal{T}$ be a table with a set of n attributes. A *partition query* Q is an aggregate query. We denote by $Q(\mathcal{T})$ the query result, and by $Q.cond$ and $Q.attr$ the attribute sets in the `WHERE` and `GROUP BY` expressions of Q, resp. The following query, denoted by Q, is an example of a partition query.

```
SELECT Aged_over_35, avg(Salary)
 FROM T
GROUP BY Aged_over_35
```

Here, $Q(\mathcal{T})$ consists of two groups (corresponds to aged under (g_1) or over (g_2) 35), $Q.cond = \emptyset$ (as no `WHERE` clause), and $Q.attr$ = {`Aged_over_35`}.

Statement. A statement is a total order *comparison* of aggregate values of two or more groups obtained by a partition query. We define a statement using a partition query and a Boolean function indicating the comparisons of some groups in $Q(\mathcal{T})$. We say that a statement holds if the conditions specified by the Boolean function are satisfied. An example of a statement associated with Q is: (S) "The average salary of respondents aged under 35 is higher than that of respondents aged over 35". This statement is defined using the following Boolean function:

$$f_Q(\mathcal{T}) = \begin{cases} 1, & \text{if } agg(g_1) - agg(g_2) \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

Query refinement. Refining a partition query allows us to determine how well the underlying data is reflected by the query. For instance, the statement S is supported by the results of the query Q, but does not reflect the fact that it is not the case for database administrators and designers. We consider typical OLAP operations *drill-down* and *slice* [8] to get partitions of the groups. Drill-down is performed by adding attributes to `Q.attr`. Slice is performed by adding conjunctions of attribute-value assignments to `Q.cond`. We define the sets of partition attributes $\mathcal{A}_{grp}$ and $\mathcal{A}_{pred}$. The $\mathcal{A}_{grp}$ attributes are used to refine the `GROUP BY` clause, and the $\mathcal{A}_{pred}$ attributes are used to refine the `WHERE` clause. Once provided the set of partition attributes by the users, OREO uses a default setting. Partition attributes that are in the same conceptual dimensions of $Q.attr$ are added to $\mathcal{A}_{grp}$. Attributes having unique values, e.g., `res ID` in Table 1, are ignored. The rest are added to $\mathcal{A}_{pred}$.

Statement Score. The score of a statement measures how well it reflects the data. To define this score, we iterate over all possible refinements of the query Q and consider the size of the sub-groups as their potential influence on the score. To this end, we define the *weight of a refinement query*, and the *support of a statement.*

The weight of a refinement query. Given $\mathcal{A}_{pred}$ and a partition query Q, a refinement expression r is an expression containing attribute-value assignments (from $\mathcal{A}_{pred}$), to be added to $Q.cond$. We denote by Q^r the refined query obtained by adding r to the `WHERE` clause of Q. A possible refinement expression to be added to the `WHERE` clause of Q is r = {`Gender = Male`}. Intuitively, the weight of a refinement is the proportion of the sub-groups that qualify. In our example, the weight of Q^r is the fraction of male respondents, i.e. $\frac{1}{2}$.

The support of a statement. Given an attribute set $A \subseteq \mathcal{A}_{grp}$, we denote by Q_A the refinement query obtained by adding the attributes in A to $Q.attr$. By adding attributes into $Q.attr$, we partition each group $g_i \subseteq Q(\mathcal{T})$ into multiple sub-groups. We then perform cross-group comparisons to compute the support. Intuitively, the support of a statement is the fraction of the sub-group comparisons that support the statement. For example, by adding the attribute set `A = {Age}` to $Q.attr$, we refine each group in Q into two sub-groups. The total number of sub-group combinations considered

for the support computation is 4. The comparisons of different age ranges that satisfy the statement are ⟨[25-34], [35-44]⟩, ⟨[25-34], [45-54]⟩ and ⟨[18-24], [45-54]⟩. Therefore, the support of the statement w.r.t. A is $\frac{3}{4} = 0.75$ (three out of four comparisons hold).

The score of a statement. The score of a statement considers all refinement queries obtained by either modifying $Q.cond$ or $Q.attr$ using the sets of partition attributes. Intuitively, the score reflects the population of sub-groups supporting the generalization. A higher score indicates a better reflection of the data. In our example, the score of S is 0.672. Note that although the generalized aggregation supports the statement, it still gets a relatively low score since a large fraction of sub-groups opposing the statement.

2.2 Problem Formulation

In OREO we address the following three problems:

Statement validation: Given a statement S, and the partition attribute sets $\mathcal{A}_{pred}$ and $\mathcal{A}_{grp}$ compute the score of S.

In case the statement score is low, the user may wish to: (i) understand which parts of the data do not "agree with the statement", and (ii) refine the statement s.t. the new refined statement better represents the data but is "as close as possible" to the original query. We thus formalize the additional two problems:

Counterargument identification: Intuitively, sub-groups (of the groups considered by the statement) that do not align with the statement reduce the statement's score. Their sizes are used to quantify the effect on the score. In this problem, we identify the set of the most general sub-groups and report the top-k of them.

Statement refinement: Given a threshold τ and a statement S s.t. $score(S) \leq \tau$, find the set of the most general statements that correspond to refinements queries of Q with a score higher than τ.

2.3 Algorithms

OREO employs the algorithms in [13] to handle the above-mentioned problems. Given a generalized statement and the sets of partition attributes, the algorithm for efficient score computation is based on a hierarchy over refinement queries and a dedicated data structure. Our algorithm uses a Hasse diagram named query refinement hierarchy (QRH) that represents a partial order over the refinement queries. Enumerating the refined sub-groups in a bottom-up fashion, allows for the reuse of computational results. The score computation algorithm traverses over all refinement queries. Therefore, identifying the counterarguments can be done alongside the score computation using extra bookkeeping. To ensure high efficiency in terms of memory consumption, to keep track of the set of counterarguments, we use the inverted index technique [9]. Finding the set of the most general alternatives via statement refinement requires computing the scores for all candidate refinement queries. A naive solution may apply a top-down breadth-first search over the QRH, utilizing the score computation algorithm to evaluate the score of each candidate query. To reduce execution times, we developed a recursive score computation method to avoid the repeated application of the score computation algorithm.

3 SYSTEM DETAILS

We have implemented OREO using Python and Flask.

System overview. As shown in Figure 1, OREO contains a user interface and three major components: dataset preparation, query builder, and statement builder. Given a data file uploaded by the data users, the dataset preparation component generates the data preview and prepares the dataset D for validation. The users can use the query builder and statement builder to construct the aggregate statement to validate (see details below). OREO then constructs the query refinement hierarchy (QRH) mentioned in Section 2.3 for efficient (score, counterarguments, and query refinement) computation. OREO uses the default settings described in Section 2.1 for the input partition attribute sets $\mathcal{A}_{pred}$ and $\mathcal{A}_{grp}$and return the statement score, top-k counterarguments and alternative statements to the system user.

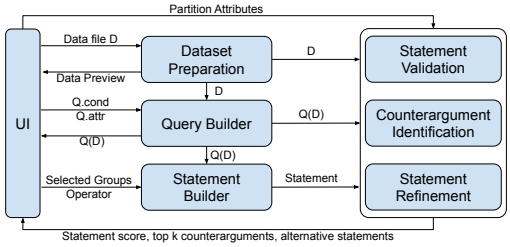


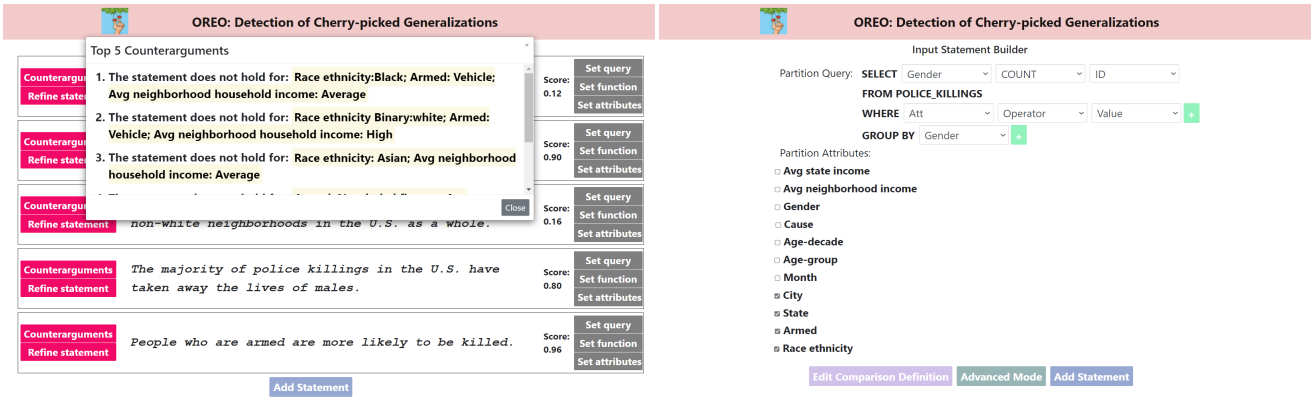
Figure 1: Overview of OREO.

User Interface. The users of OREO can either choose from the preloaded datasets or upload their own data files. To identify the statement for validation, OREO supports two input configurations.

Input: preloaded statements. As shown in Figure 2(a), the users will first be able to explore some example statements derived from real-world sources and examine their validation results. By clicking on the gray buttons to the right of the selected statement, the users can modify the corresponding partition query, Boolean function, and partition attributes of the statement.

Input: manually-defined statements. Interested users could insert their own statements by clicking on the "Add Statement" button (see the bottom part of Figure 2(a)). The users can express their statement using the statement builder shown in Figure 2(b). As a generalized statement is derived by an aggregate query, the users first use the query builder to specify the components of the partition query, i.e. $Q.attr$, $Q.cond$, the aggregate function, and the target attribute. Then the users click on the "Edit Comparison Definition" button to specify the boolean function and the groups in the aggregation results that they what to compare. They can also input more complex partition queries in SQL format using the "Advanced Mode" button. Then the users could choose the partition attributes for query refinement from the below attribute list.

Output: validation results. After specifying the statement to be investigated, OREO displays the statement scores computed by our algorithm (as shown in Figure 2(a)). The users will then be invited to further investigate the validation results by clicking on the pink buttons to the left of the statement, OREO enables one to understand the statement's score better (e.g., by identifying which parts of the data do not "agree with the statement"). In more detail, by clicking on the "Counterarguments" button, OREO displays (in a popup window) the top 5 counterarguments associated with the largest sub-groups that do not align with the selected statement. The top 5 counterarguments for the 4-th statement are shown in Figure 2(a). For instance, an example counterargument for this statement is that



(a) Top 5 counterarguments for the 4-th statement.

(b) Statement builder.

Figure 2: UI of OREO.

there are more female victims than males for black victims with vehicles in neighborhoods with an average household income. The participants can also generate alternative higher-scored statements for poorly stated statements by clicking on the "Refine Statement" button. To that end, OREO requires inserting a threshold. It then performs the statement refinement task (as described in Section 2.2) for the selected statement to obtain alternatives whose score is higher than the threshold.

4 DEMONSTRATION PLAN

We demonstrate the operation of OREO over three real-world datasets, which include multiple attributes and can be associated with real-life statements: (1) **Stack Overflow** (SO) dataset: SO's annual developer survey [5] is a dataset that has more than 98K records containing information such as the developers' age, gender, ethnicity, and income. (2) **Police Killings** (PK) dataset [1], contains information regarding people killed by police and other law enforcement agencies in the United States. The attributes detailed people demographics (e.g., age, gender, ethnicity) as well as details of the cases (e.g., cause of death). (3) **Academia dataset**. We use the dataset of the academic staff at a well-known university. The dataset contains information such as gender, academic rank, and research direction (art/life/science/law), of over 1.2K academics hired by the university in the period 1990 – 2020. We derive the preloaded statements from several sources, such as Stack Overflow user reports [5], news and media websites, including The Guardian [2] and FiveThirtyEight [7]. These statements represent insights about technology trends (SO) or expose problems of contemporary society (PK). For the Academia dataset, the statements were generated by real-life analysts investigating gender gaps in university hiring.

We next use the police killings (PK) dataset and the example statement "The majority of police killings in the U.S. have taken away the lives of males." to demonstrate how to interact with OREO.

- Select the "police_killings.csv" as the dataset for validation.
- Use the statement builder to configure the statement for validation. There is no $Q.cond$ in the statement so we leave the WHERE clause blank. For the $Q.attr$, we select the *Gender* attribute from the drop-down menu. For the aggregate function and target attribute, we consider the COUNT aggregation on the id attribute. We further specify the boolean function as Count(male) > Count(female) by clicking the "Edit Comparison Definition" button.
- Select the partition attributes from the list. In this case, we consider the *City*, *State*, *Armed* and *Race ethnicity* as the partition attributes.
- The evaluation results will be the same as the 4-th preloaded statement shown in Figure 2(a). We can explore the counterarguments and alternatives by clicking the pink buttons.

ACKNOWLEDGMENTS

This research was supported in part by NSF under grants 1741022 and 1934565 and was partially funded by the Israel Science Foundation, the Binational US-Israel Science Foundation, and the Tel Aviv University Center for AI and Data Science.

REFERENCES

[1] 2015. Police Killings Dataset. https://github.com/fivethirtyeight/data/tree/master/police-killings.
[2] 2016. TheGuardian.com. https://www.theguardian.com/us-news/series/counted-us-police-killings.
[3] 2019. What happens to developers once they reach 35? https://blog.pitchme.co/2019/10/29/what-happens-to-developers-once-they-reach-35/.
[4] 2019. What to know about ADHD misdiagnosis. https://www.medicalnewstoday.com/articles/325595#age-related-factors.
[5] 2020. Stack Overflow developer survey. https://insights.stackoverflow.com/survey.
[6] Abolfazl Asudeh, Hosagrahar Visvesvaraya Jagadish, You Wu, and Cong Yu. 2020. On detecting cherry-picked trendlines. *PVLDB* 13, 6 (2020), 939–952.
[7] Ben Casselman. 2015. Where Police Have Killed Americans In 2015. https://fivethirtyeight.com/features/where-police-have-killed-americans-in-2015/.
[8] Surajit Chaudhuri and Umeshwar Dayal. 1997. An overview of data warehousing and OLAP technology. *SIGMOD* 26, 1 (1997), 65–74.
[9] Doug Cutting and Jan Pedersen. 1989. Optimization for dynamic inverted index maintenance. In *SIGIR*. ACM, 405–411.
[10] Saehan Jo, Immanuel Trummer, Weicheng Yu, Xuezhi Wang, Cong Yu, Daniel Liu, and Niyati Mehta. 2019. Verifying text summaries of relational data sets. In *SIGMOD*. ACM, 299–316.
[11] Georgios Karagiannis, Mohammed Saeed, Paolo Papotti, and Immanuel Trummer. 2020. Scrutinizer: A Mixed-Initiative Approach to Large-Scale, Data-Driven Claim Verification. *PVLDB* 13, 12 (2020), 2508–2521.
[12] Georgios Karagiannis, Mohammed Saeed, Paolo Papotti, and Immanuel Trummer. 2020. Scrutinizer: fact checking statistical claims. *PVLDB* 13, 12 (2020), 2965–2968.
[13] Yin Lin, Brit Youngmann, Yuval Moskovitch, H. V. Jagadish, and Tova Milo. 2022. On Detecting Cherry-Picked Generalizations. *PVLDB* 15, 1 (2022), 59–71.
[14] You Wu, Pankaj K Agarwal, Chengkai Li, Jun Yang, and Cong Yu. 2014. Toward computational fact-checking. *PVLDB* 7, 7 (2014), 589–600.
[15] You Wu, Pankaj K Agarwal, Chengkai Li, Jun Yang, and Cong Yu. 2017. Computational fact checking through query perturbations. *TODS* 42, 1 (2017), 1–41.