Selecting Subgraph Matching Algorithm via Machine Learning [Scalable Data Science]

Genryu Kuraya
The University of Osaka
Suita, Japan
kuraya.genryu@ist.osaka-u.ac.jp

Skitsas Konstantinos
Aarhus University
Aarhus, Denmark
skitsas@cs.au.dk

Panagiotis Karras
University of Copenhagen
Copenhagen, Denmark
piekarras@gmail.com

Daichi Amagata
The University of Osaka
Suita, Japan
amagata.daichi@ist.osaka-u.ac.jp

Yuya Sasaki
The University of Osaka
Suita, Japan
sasaki@ist.osaka-u.ac.jp

ABSTRACT

Subgraph matching is a fundamental problem in graph analysis that seeks all instances (or *embeddings*) of a *query subgraph* within a larger *data graph*. Numerous subgraph matching algorithms have been developed for efficient processing. However, the best-performing algorithm differs across query and data graphs. A previous study proposed manually designed rule-based models for selecting the algorithm to use depending on query and data characteristics. However, this rule-based model becomes ineffective on unseen data graphs. In this paper, we propose a machine-learning-based framework to select a subgraph matching algorithm to use for the sake of efficiency, based on the query and the data. Our framework learns from observations of the performance of subgraph matching algorithms on different data. It comprises two key components: a *labeling strategy*, which adds labels to experimental results, and a *featurizer*, which extracts well-designed features from datasets and queries. Our experiments show that our framework selects high-performing subgraph matching algorithms and improves throughput of embeddings per second by up to 76% over baselines.

PVLDB Reference Format:
Genryu Kuraya, Skitsas Konstantinos, Panagiotis Karras, Daichi Amagata, and Yuya Sasaki. Selecting Subgraph Matching Algorithm via Machine Learning [Scalable Data Science]. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/GenryuK/Selecting-Subgraph-Matching-Algorithm-via-Machine-Learning.git.

1 INTRODUCTION

Subgraph matching is a fundamental problem in graph analysis that seeks identifies occurrences (i.e., *embeddings*) of a given *query graph* within a much larger *data graph*. This task has a wide variety of applications such as fraud detection [19], social network analysis [5], bioinformatics [3], and knowledge graph analysis [11, 18]. However, it is computationally challenging and known to be $\mathbb{NP}$-hard [8]. Numerous algorithms have thus been proposed over decades to accelerate finding subgraphs [4, 6, 10, 27]. However, despite those efforts, there is no single *algorithm of choice* that performs reliably well across problem instances. Each algorithm has pros and cons depending on the query and data graphs. For instance, GQL [9] performs well on dense graphs but poorly on sparse ones, whereas RI [4] presents the opposite behavior. Previous benchmarking studies have empirically compared existing algorithms across various data graphs and concluded that there is no clear winner [14, 25, 31].

Figure 1 shows a heatmap of throughput in terms of *embeddings per second* (EPS) [31] over sixty algorithms on seven real-world data sets. Figure 2 plots the average EPS for each algorithm rank on each data set. These results show that the best-performing algorithm is highly data-dependent and efficiency decreases drastically when selecting an unsuitable algorithm for given query and data.

Given this predicament, there is a need for a way to select the best subgraph matching algorithm, given a query and a data graph. Prior studies [25, 31] have proposed manual, rule-based models to select algorithms based on features of the query and data graphs according to empirical performance data. However, these models lose effectiveness when applied to query and data graphs that differ from those in their experiments. There is thus a need to robustly select the algorithm to use for a given query and data.

In this paper, we propose a novel framework that employs machine learning to automatically select and generate the subgraph matching algorithm to employ for a given problem instance, out of a realm of possibilities. We train a machine learning model on empirical performance data, which include quadruples of a *data graph*, a *query graph*, an *algorithm*, and the associated *performance*. The key components of our framework are a *labeling strategy*, which builds training data by adding labels to empirical performance data, and a *featurizer*, which extracts well-designed features from query and data graphs. We test three labeling strategies for assigning target labels, defined in terms of how they select the algorithm to which they assign a positive label: Top-X, Inc-Y, and Weight. The Top-X strategy selects the top-X performing algorithms for each query and data graph pair. The Inc-Y strategy selects the algorithms whose performance is within a Y ratio of the best-performing algorithms. The Weight strategy selects algorithms based on their performance compared to the top-performing one. The featurizer extracts 13

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

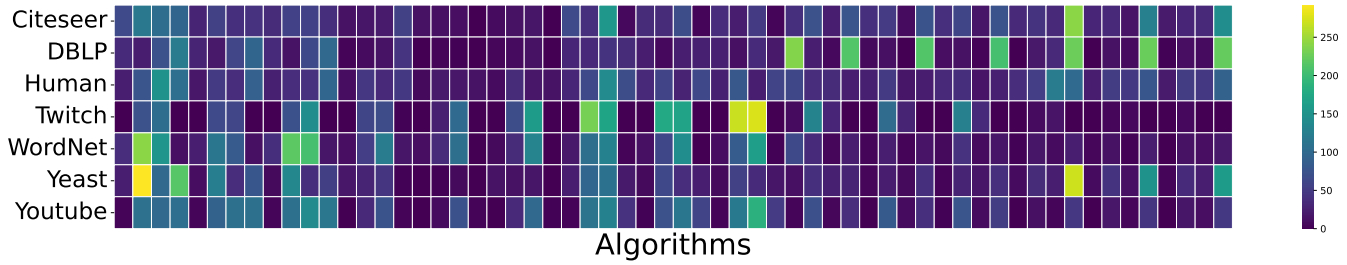


Figure 1: A heatmap illustrating the algorithm that achieves the highest EPS in each dataset.

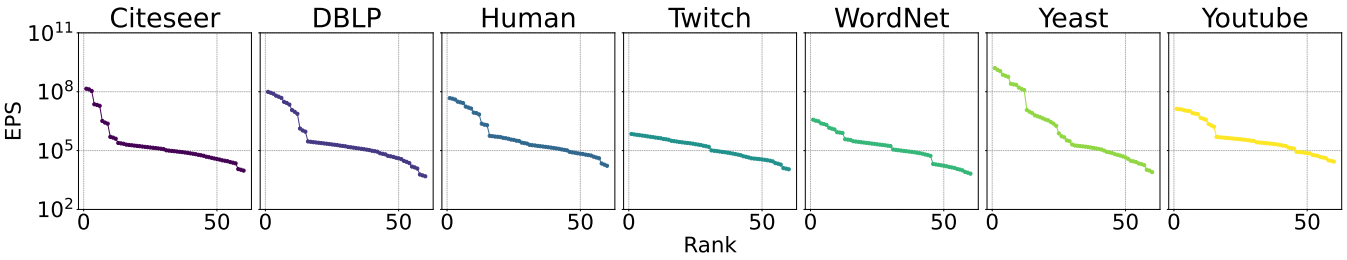


Figure 2: Average EPS per rank; the average EPS on rank i is the EPS we obtain if we always select the algorithm ranking i for each query among 60 algorithms.

features from query and data graphs, such as density and diameter. Put together, the labeling strategy and featurizer enable training a machine learning model on a classification task that predicts the algorithm to employ, as a *class*, for given query and data features. In the inference phase, the trained model selects the algorithm predicted to perform best on a given query and data graph, and executes subgraph matching using the selected algorithm.

In our experimental study, we evaluate sixty subgraph matching algorithms on seven real-world data graphs. Our selection framework improves performance by up to 76% in EPS over baselines, and 90% in mean reciprocal rank (MRR) in terms of selecting the best performer. We also establish the robustness of our framework even when training on data graphs and query types different from the ones we test on.

We outline our contributions as follows:

- **Novel ML4DB problem:** We leverage machine learning to select the subgraph matching algorithm to employ; to our knowledge, no prior work has addressed this task.
- **Novel framework:** We propose a comprehensive framework that selects a subgraph matching algorithm via machine learning, using a labeling strategy and a feature extraction module.
- **Empirical validation:** Extensive experiments on seven real-world datasets demonstrate that our framework consistently outperforms baselines in various scenarios.

2 PRELIMINARIES

Here we present core definitions and related work.

2.1 Graph and subgraph matching

We define $g = (V(g), E(g), L_g)$ to be a labeled undirected graph, where $V(g)$ is a set of vertices and $E(g)$ is a set of edges. Edge $e(v, v') \in E(g)$ indicates that $v, v' \in V(g)$ are connected in g. L_g is a mapping from each vertex to the set of vertex labels Σ.

Given a query graph $q = (V(q), E(q), L_q)$ and a data graph $G = (V(G), E(G), L_G)$, an *embedding* of q in G is a mapping $M : V(q) \rightarrow V(G)$ such that: (i) M is *injective*, i.e., if $u \neq u' \in V(q)$, then $M(u) \neq M(u')$, (ii) $L_q(u) = L_G(M(u)), \forall u \in V(q)$, and (iii) $e(M(u), M(u')) \in E(G), \forall e(u, u') \in E(q)$. If there exists an embedding of q in G, then q is *subgraph isomorphic* to G. The *subgraph matching* problem calls to find all embeddings of query graph q in data graph G. In most cases, the data graph G is significantly larger than the query graph q.

2.2 Related Work

Subgraph Matching Algorithms. Subgraph matching algorithms feature three stages: filtering, ordering, and enumeration [25, 31]. *Filtering* narrows the search space down to the vertices and edges in the data graph that could potentially match the query graph based on structural and label information. *Ordering* determines a guiding sequence by which vertices in the query graph are efficiently mapped to the data graph. *Enumeration* outputs all embeddings of the query graph.

Numerous subgraph matching algorithms have been proposed. LDF [27] filters unnecessary vertices in data graphs based on vertex labels and degrees. NLF [32] enhances filtering by evaluating neighborhood labels. GQL [9] determines an efficient matching order based on candidate size, and LFTJ [28] enhances the worst-case computational complexity with respect to a relational database. RI [4] uses an ordering technique that selects vertices most connected to the already ordered vertices set. DPiso [7] converts the query graph into a directed acyclic graph (DAG) for dynamic filtering and utilizes *failing sets*, i.e., unmatchable subsets of the query, to prune the search space. RM [26] treats subgraph matching as a problem on relational database tables. VEQ [10] determines a matching order by vertex groups and dynamically prunes the search space using

equivalences. KSS [30] reduces the search space by decomposing the query graph into a *kernel* and a *shell*, confining the main computation to the kernel. PILOS [23] adopts a strong spectral filtering technique based on Laplacian matrices and the *interlacing theorem* as an algebraic constraints.

Subgraph Matching with Machine Learning. Several studies [12, 13, 15, 16, 29] use machine learning for subgraph matching. NeuroMatch [16] performs binary classification to determine whether a query graph exists in the data graph. Sub-GMN [13] and AED-Net [12] identify a single embedding of query graph on data graph. QVO [29] and RSM [15] leverage reinforcement learning, QVO [29] to improve the effectiveness of the ordering step, RSM [15] to determine the order of candidate generation in the enumeration step.

Benchmarking. Several benchmarking studies evaluate the performance of algorithms to understand their characteristics [14, 25, 31]. Such studies have established that the best-performing algorithm depends strongly on the query and data graphs. Lee et al. [14] re-implemented existing algorithms under a common environment and evaluated each algorithm. Sun et al. [25] proposed the decomposition of algorithms into filtering, ordering, enumeration, and optimization steps and analyze the impact of each step on the performance. Further, Zhang et al. [31] generate new algorithms by recombining techniques for each stage of existing methods and show that some new recombined algorithms outperform the original ones. In this work, we study how to select recombined algorithms based on the query and data characteristics.

3 MOTIVATION AND PROBLEM DEFINITION

Figure 1 presents the performance distribution of sixty algorithms in seven datasets by the EPS performance measure, which indicates the average throughput of identified embeddings per second [31]. We execute 2 800 queries for each dataset (see Section 5 for more details). From the figure, we observe that the most efficient algorithm varies significantly across datasets. For example, DBLP and WordNet present largely deviating results on the best-performing algorithm. On DBLP, the right-hand-side algorithms in Figure 1 tend to perform best, whereas on WordNetthe left-hand-side algorithms have an advantage. On the other hand, some data, for example, WordNet and Youtube, exhibit similar trends.

Besides, Figure 2 shows that EPS does not grow linearly vs. rank. For example, in Yeast, the EPS difference between the top-ranked and the lower-ranked algorithms reaches several orders of magnitude, whereas in Twitchthat difference is more modest. In addition, the EPS is highly different even among top-ranked algorithms, for instance the algorithms ranked third and fourth on Citeseer. These results suggest that it may be always critical to select the top algorithm, yet there are instances in which even the algorithm ranked fifth may suffer a significant loss over the top one. These results indicate that predicting the most efficient algorithm given a query and a data graph is a challenging task. We thus conjecture that it may be practical to select algorithms based on *empirical performance data* obtained from experimental results, which we define as follows:

Definition 1 (Empirical performance data). *Given a set $\mathcal{A}$ of algorithms, empirical performance data includes a set of quadruples <G, q, algorithm, performance measure> for any algorithm in $\mathcal{A}$, where G identifies a data graph and q a query.*

While we use EPS [31] as a performance measure, any other measure, such as runtime for a task, would be applicable. We assume that a performance measurement is available for each data graph, query, and algorithm in $\mathcal{A}$, hence we know the best-performing algorithm for a given q on G. We this define the problem we address as follows:

Problem 1 (Subgraph Matching Algorithm Selection). *Given query graph q, a data graph G, a set $\mathcal{A}$ of subgraph matching algorithms the* subgraph matching algorithm selection *problem calls to select the best-performing algorithm from $\mathcal{A}$ for q and G.*

A straightforward approach to the problem select an algorithm that usually achieves high performance based on the empirical performance data. For example, one may select the subgraph matching algorithm that often achieves the best performance for either all data graphs or the same data graph. However, such an approach may not select the best algorithm for each query, as it does not consider the query graph.

4 OUR FRAMEWORK

Here we present our framework that selects a subgraph matching algorithm with the highest predicted performance among a given set of algorithms via machine learning and performs subgraph matching with the selected algorithm. Figure 3 presents an overview. We perform a classification task, in which each class label represents an algorithm, the inputs are features of the query and data graphs. The model learns to predict the best-performing algorithm given the features of the query and data graphs.

In the training phase, we extract features from the query and data graphs and assign labels corresponding to algorithm performance according to our labeling strategies. In the inference phase, we extract features from the given query and data graphs and input them into the trained model. The model infers an algorithm expected to achieve the highest performance, so our framework executes subgraph matching using the selected algorithm.

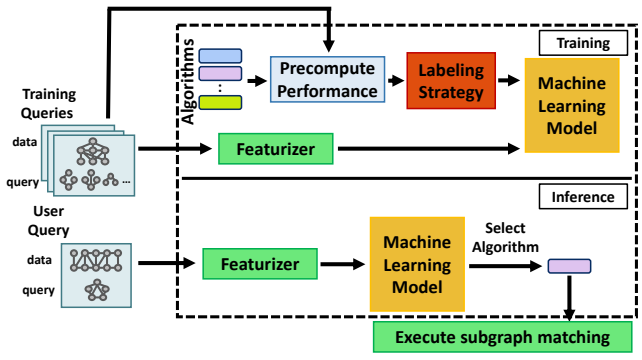


Figure 3: An overview of framework for subgraph matching via machine learning.

4.1 Labeling Strategies

To render empirical performance data usable for model training, we assign labels to them. When labeling algorithms for a given query, we assign labels to multiple algorithms with high performance instead of only selecting the single algorithm with the highest performance, based on our empirical analysis in Section 3. Mathematically, we aim to learn a performance function $f: G \times q \rightarrow \mathbb{R}^{|\mathcal{A}|}$. Thus, the size of the training data is the number of queries.

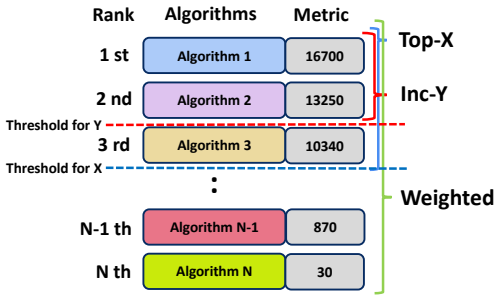


Figure 4: Examples of three labeling strategies.

We propose three labeling strategies as follows:

- **Top-X:** Assign positive labels to the top-X performing algorithms.
- **Inc-Y:** Assign positive labels to algorithms performing at least a ratio Y of the best performance.
- **Weight:** Assign positive labels to algorithms based on weighted values, computed as their relative performance.

Figure 4 illustrates how our labeling strategies work with an example. By the Top-X strategy with $X = 3$, we assign label 1 to the top-3 algorithms and 0 to others. By the Inc-Y strategy with $Y = 0.7$, we assign label 1 to the algorithms whose EPS are higher than $0.7 \cdot 16,700 = 11,690$ and 0 to others. By Weight strategy, we assign labels 1.0, 0.79, 0.62, 0.05, and 0.002 to the 1st, 2nd, 3rd, $(N-1)$th, and Nth algorithms, respectively.

4.2 Featurizer

Featurizer extracts 13 structural features from query and data graphs that represent their characteristics. These features include the numbers of vertices in the query and data graphs (i.e., $|V(q)|$ and $|V(G)|$), the number of edges in the query graph (i.e., $|E(q)|$), the average degree of the query graph, the diameter of the query graph, the core numbers of query and data graphs [22], the density of query and data graphs (i.e., $\frac{2|E(q)|}{|V(q)|}$ and $\frac{2|E(G)|}{|V(G)|}$), the tree width [20], label set size (i.e., $|\Sigma|$), and *candidate size* of the data graph, and the *label ratio* between the query and data graphs. We define the last two measures, which are not standard measures in graph analysis.

Candidate size is the average number of candidate vertices involved in subgraph matching, as computed by the LDF filtering algorithm [27]: $\frac{1}{|V(q)|} \sum_{u \in V(q)} |C(u)|$ where $|C(u)|$ is the number of candidates vertices in G for query vertex u. *Label ratio*, which we propose, is the dot product of the label distributions in the query graph and the data graph: $\frac{\sum_{\ell \in \Sigma} |V(q)_\ell| \cdot |V(G)_\ell|}{|V(q)| \cdot |V(G)|}$, where $V(q)_\ell$ and $V(G)_\ell$ denote the sets of vertices with label ℓ in q and G, respectively. Four of these features, namely the density, label set size of the data graph, candidate size, and tree width, appear in the rules proposed by prior work [31]. We add the other nine features, as we consider them essential for selecting subgraph matching algorithms.

4.3 Model Training

Here we explain how we train the machine learning model. We use the standard cross-entropy loss [21] as follows:

$$\mathcal{L} = -\frac{1}{|Q|} \sum_{q \in Q} \sum_{i=1}^{|\mathcal{A}|} y_{q,i} \log \frac{\exp(x_{q,i})}{\sum_{j=1}^{|\mathcal{A}|} \exp(x_{q,j})} \quad (1)$$

where $x_{q,c}, c \in [1, |\mathcal{A}|]$, denotes the output by the machine learning model and $y_{q,c}$ denotes the label for class c in query q. Q denotes the set of queries in training and $|Q|$ denotes their number.

5 EXPERIMENTS

To evaluate the effectiveness of our methods, we conduct experiments to clarify how to efficiently train machine learning models to select the best algorithms and whether the trained models are robust.

5.1 Experimental Setup

Algorithms and Experimental Environment. We consider the methods listed in Table 1 for each stage of the subgraph matching process and produce 60 algorithms as their $5 \cdot 3 \cdot 4$ combinations. PILOS [23] is a recently proposed filtering method, while other algorithms have been recommended in a prior benchmarking study [31]. We use a publicly available source code[1] [31]. Note that the enumeration method of RM does not work for some queries, so we remove RM in those cases.

Table 1: Subgraph matching algorithm. We combine methods from each stage.

Stages	Methods
Filtering	LDF [27], NLF [32], DPiso [7], VEQ [10], PILOS [23]
Ordering	RI [4], RM [26], GQL [9]
Enumeration	EXPLORE [27], LFTJ [28], KSS [30], VEQ [10]

The subgraph matching algorithms are entirely implemented in C++, whereas we developed all machine learning components in Python3. We compiled the C++ code using CMake version 3.22.1 and g++ version 11.4.0. We ran experiments on a server running Ubuntu 22.04.3 LTS, equipped with an Intel(R) Xeon(R) Gold 6248R CPU and 755 GB of RAM.

To ensure the reliability of our results and mitigate the effects of random initialization, we conducted all experiments using five different seeds. The performance values we report represent the average across these five runs.

Datasets. We use the seven data graphs presented in Table 2; the same datasets have been widely used in previous studies (e.g.,

[1] https://github.com/JackChuengQAQ/SubgraphMatchingSurvey

[25, 31]). Since Twitch originally has no labels, we assign labels uniformly at random following existing works [26].

Query Sets. We generate two types of query sets; induced graphs Q_I and star graphs Q_S. To generate a Q_I query graph, we start from one vertex in the data graph, extend edges to its connected neighbors, and then move to one of those neighbor vertices and repeat this process until we reach the desired number of vertices. Thereafter, we extract all edges that connect the extracted vertices. To generate a Q_S query graph, we start from one vertex in the data graph, extending edges to its connected neighbors, and then move to a neighbor vertex that *has not been extracted* and repeat this process until we reach the desired number of vertices. We extract queries of seven different sizes, where size $i \in \{4, 8, 16, 32, 64, 96, 128\}$ is the number of vertices; we indicate as Q_i the set of queries of size i.

We prepare 200 queries of each of Q_{Ii} and Q_{Si} from each dataset (i.e., 2, 800 across all datasets), yield a total of 19, 600 (i.e., 2,800 × 7) query-data pairs. We remove queries for which all algorithms fail to find any embeddings.

Table 2: Dataset Statistics Used in Experiments. k denotes the core number of the graph.

Dataset	Name	$\|V\|$	$\|E\|$	k	$\|\Sigma\|$	Type
Citeseer[2]	cs	3,279	4,552	7	6	Citation
DBLP[3]	db	317,080	1,049,866	113	14	Collab
Human[3]	hm	4,674	86,282	148	43	Protein
Twitch[4]	tw	168,114	6,797,557	149	45	Social
WordNet[3]	wn	146,005	656,999	31	4	Lexical
Yeast[3]	ys	2,361	7,182	10	70	Protein
Youtube[3]	yt	1,134,890	2,987,624	51	23	Social

Baselines. We compare our methods against the following baselines:

- **Oracle**, which selects the algorithm that yields the highest EPS for each query. Since the results obtained by Oracle represent the best values we can get, we compare the difference between Oracle and other methods.
- **DatasetFast**, which selects the algorithm that most frequently achieves the highest EPS on a given dataset.
- **AllFast**, which selects the algorithm that most frequently achieves the highest EPS across all datasets.
- **RecAlgo** [31], which selects methods for filtering, ordering, and enumeration by a manually designed rule-based model based on the characteristics of the graph.

ML Model and Hyper-parameter. We employ a 2-layer MLP as our machine learning model due to its ease of implementation and ability to train quickly. We apply a random 6 : 2 : 2 split to the query set for training, validation, and testing with respect to each data graph.

For hyper-parameter tuning, the tuning ranges are [5e-4, 1e-3, 5e-3, 1e-2, 5e-2] for learning rate, [0.0, 0.1, 0.2, 0.3, 0.4, 0.5] for drop out rate, [0.0, 1e-5, 1e-4, 1e-3, 1e-2] for weight decay, and [16, 32, 64, 128] for unit size in hidden layer. [2, 17, 24]. Each labeling strategy has its own unique hyper-parameters: for Top-X, we choose the value of X from the set [1, 2, 3, 4, 5, 6, 7]; for Inc-Y, we choose the value of Y is chosen from [1.0, 0.95, 0.9, 0.85, 0.8, 0.75, 0.7, 0.65, 0.6, 0.55, 0.5]; for Weight, we choose the value of α from [1.0, 0.95, 0.9]. We use Optuna [1] to minimize the validation loss.

Metrics. We report the average EPS (embedding per second) [31] and MRR (mean reciprocal rank) over five random splits. EPS indicates how many embeddings are processed per unit of runtime, where runtime includes filtering, ordering, and enumeration, and, in addition, for our methods and RecAlgo, also time to extract features from query graphs and inference time. We set the timeout of enumeration to 10 seconds. MRR evaluates how close a method comes to selecting the true best-performing algorithms, calculated as $MRR = \frac{1}{|Q|} \sum_{q \in Q} \frac{1}{rank_q}$, where $rank_q$ denotes the true rank of the algorithm that a method selects for query q.

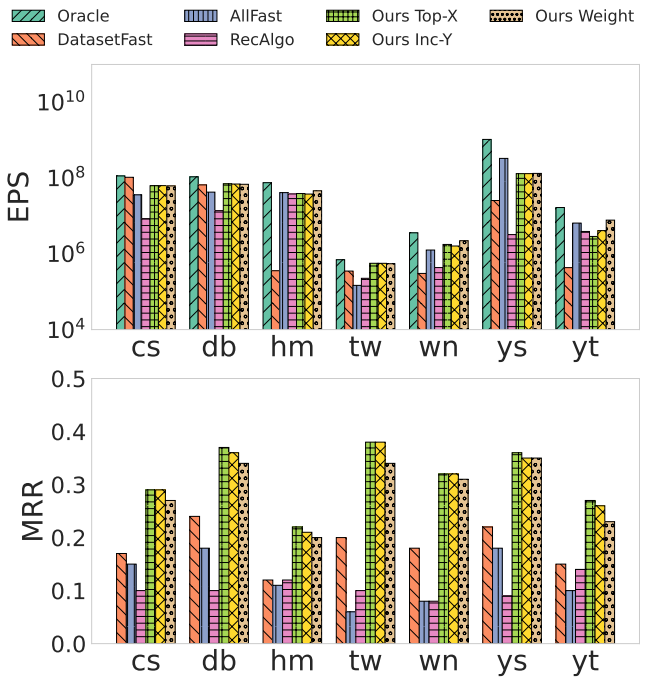


Figure 5: Overview comparison: EPS (top) and MRR (bottom).

5.2 Comparison with baselines

We compare the performance of our methods vs. baselines. Figure 5 shows EPS and MRR for each method. Our methods outperform baselines on five datasets (i.e., db, hm, tw, wn, and yt) in EPS. In particular, in wn, Weight achieves approximately twice the EPS of AllFast. In cs and ys, our methods achieve the second highest EPS and are still close to Oracle in cs. In ys, the EPS gaps between Oracle and our methods are larger than on other datasets. This is because ys contains more lightweight queries than other datasets, which originally finish in about 0.1 ms. For such queries, the specific cost of our methods for model inference and feature extraction, which is around 1 ms, may result in an overhead. Thus, even if a method selects highly-ranked algorithms, its gap from Oracle becomes large

[2] https://networkrepository.com/citeseer.php
[3] https://github.com/RapidsAtHKUST/SubgraphMatching
[4] https://snap.stanford.edu/data/twitch_gamers.html

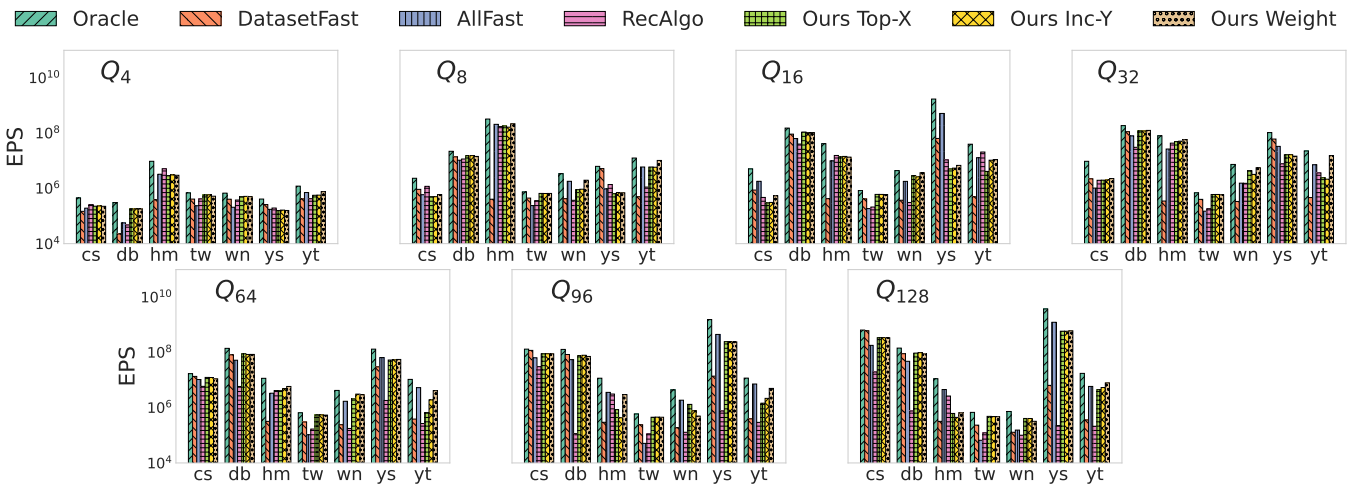
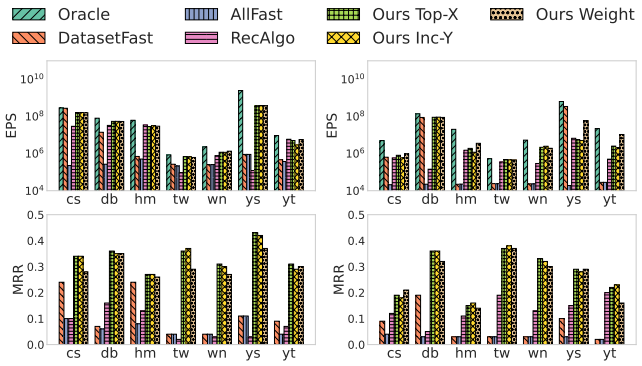


Figure 6: EPS on each query size.

if it fails to select the top-ranked algorithm. Among our methods, Weight achieves relatively higher EPS than Top-X and Inc-Y.



(a) Q_I **(b)** Q_S

Figure 7: EPS and MRR on each query type.

On MRR, our methods outperform baselines on all datasets. This result shows that our methods consistently select highly-ranked algorithms. In tw, our methods achieve an MRR score of almost 0.4, which indicates that our methods select the second or third best algorithms on average. In ys, even though AllFast achieves high EPS performance, its MRR value is not so high, around 0.2. This indicates that AllFast selects top-rank algorithms for some queries that have significantly higher EPS, while it fails on others.

Figure 6 shows the EPS for each query size per dataset. In some datasets (e.g., cs, db, ys and yt), the EPS of Oracle increases with query size, while in tw, the average EPS is stable across query sizes. This result suggests that the performance of baselines, in particular DatasetFast, worsens as query size increases, while our methods still achieve high performance. However, on ys for Q_{16}, where the EPS of Oracle is around one billion, our methods fail to approach Oracle as AllFast does. This result shows the reason why AllFast is strong for ys.

Figure 7 plots the EPS and MRR for each query type. Both results show that our methods outperform baselines on most datasets. With

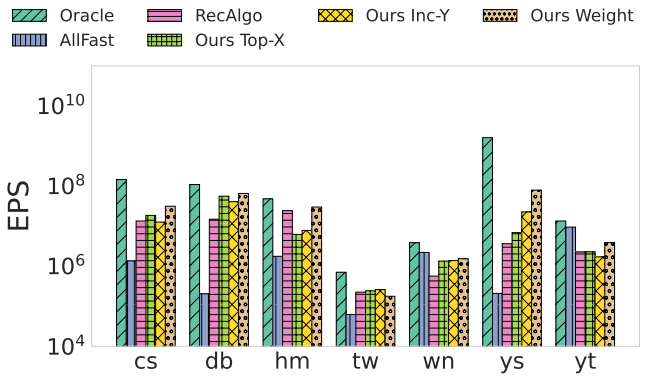


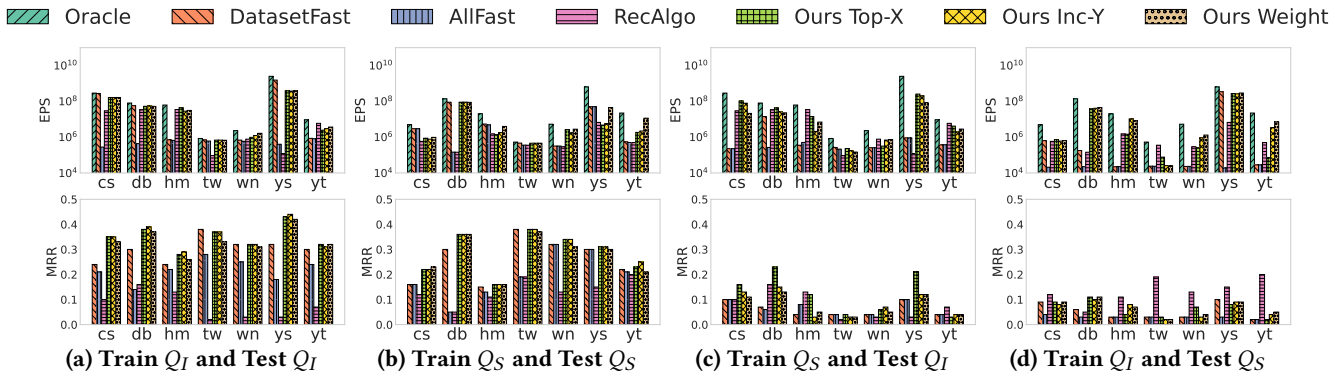
Figure 8: EPS on unseen datasets.

query set Q_I, our methods outperform baselines on four datasets, db, tw, wn, and ys, while with query set Q_S, our methods outperform baselines on six datasets, other than ys. These results indicate that, if one prepares sufficient training queries (e.g., various structures such as Q_I and Q_S), our methods achieve high performance on a structural biased query sets.

Overall, we find that our methods outperform baselines. In the generic scenario, our methods achieve up to approximately twice ESP than baselines. In the restricted scenario (e.g., where we use queries of only one type as test set), our methods still achieve higher performance than baselines.

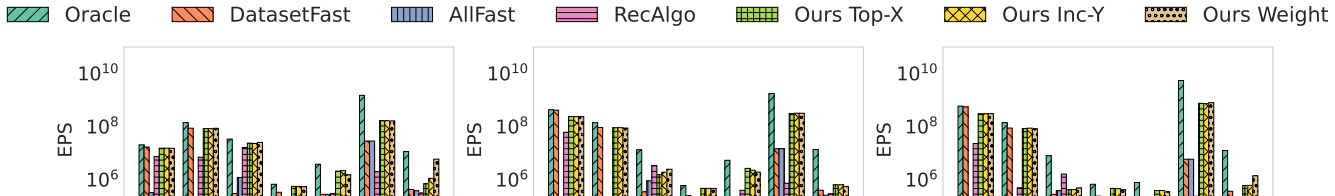
5.3 Evaluation on unseen datasets

We now evaluate the performance of our methods on unseen datasets to test their generalizability beyond training data. We train and validate on queries in six of the seven datasets, and test on queries from the remaining dataset. Thus we cannot evaluate DatasetFast because there are no results in the test datasets. Figure 8 shows the EPS of each method. Our methods outperform the baselines by up to two orders of magnitude on five datasets, cs, db, hm, tw, and ys.



(a) Train Q_I and Test Q_I **(b) Train Q_S and Test Q_S** **(c) Train Q_S and Test Q_I** **(d) Train Q_I and Test Q_S**

Figure 9: The performance of different patterns of training and test queries.



(a) Test Q_{64} **(b) Test Q_{96}** **(c) Test Q_{128}**

Figure 10: EPS on unseen query size; removing a specific size of queries from training and test the performance of the queries.

In ys, baselines perform approximately three orders of magnitude worse than the Oracle, while our methods achieve 22 times higher performance than the baselines. These results indicate that our methods achieve high performance against queries from unseen datasets; we conclude that our methods achieve strong robustness.

5.4 Effect of training query set

We also evaluate the effect of the training query set to validate the robustness of our framework. Figure 9 shows the EPS for different patterns of training and test query sets for Q_I and Q_S. Figures 9 (a) and (b) show the results in consistent training setting, whereas Figures 9 (c) and (d) show the results in inconsistent training setting. In the consistent training setting, our methods outperform baselines on two or three datasets. Although the baselines achieve higher performance on other datasets, the difference between them and the performance of our methods is slight in some cases. In the inconsistent training setting, our methods outperform baselines on more than half datasets. In those scenarios, DatasetFast and AllFast do not perform well compared with their results in consistent setting. In particular, on cs, the EPS of DatasetFast drops by approximately three orders of magnitude while our methods maintain a high performance. In addition, as Figures 9 (c) and (d) show, our methods outperform the baselines on more datasets when training on induced graphs, indicating that induced queries may be more effective than star queries in training.

Next, we evaluate robustness in the size of query graphs. We remove one of the query sizes from the training data and then evaluate the performance of those queries. We test on queries whose size are 64, 96, and 128. Figure 10 shows the EPS for each query size. Our methods outperform baselines on most cases. For example, in Q_{64}, our methods achieve up to nearly 10 times higher EPS than baselines, and in Q_{128} on ys, our methods outperform baselines by two orders of magnitude. These results reconfirm that our methods are robust on query sizes beyond those used during training.

Overall, our methods maintain high performance even though the test set only contains unseen structures and sizes of queries. Thus, our methods are robust to training sets of queries with different structures and sizes.

5.5 Effect of training data size

We now examine the performance of our methods as the size of the training data decreases. By default, the training set contains about 1600 queries per dataset. We reduce this number to 800, 400, 200 and 10, while keeping the same test data. Figure 11 plots MRR vs. training data size. The values of DatasetFast and AllFast also change depending on training set size because they decide an algorithm based on training set, while RecAlgo does not change as it decides algorithm by pre-defined rules.

The results show that the performance of our methods does not change significantly even as we reduce the training set size to 200. In hm, the performance drop is at most only 5% as the training set size changes from 1600 to 200. When we reduce the training set size to 10, performance degrades across all datasets, yet our methods still outperform the baselines in most cases. In particular, in tw and wn, the performance drop from training set size 200 to 10 is at most 43%. Among our methods, Inc-Y achieves a relatively high MRR. On the

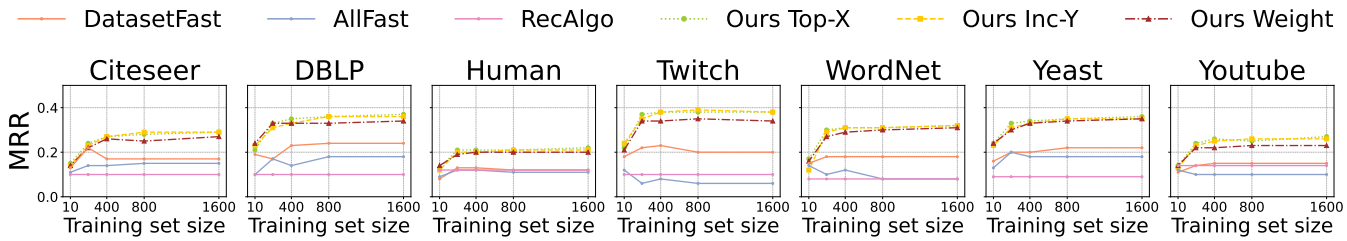


Figure 11: Impact of training data size on MRR across datasets.

other hand, DatasetFast and AllFast show different trends in MRR depending on the training size. DatasetFast looses performance when the training size is reduced to 10, whereas AllFast shows a slight improvement on tw, wn, and yt, due to an accidental fit of the reduced training set. DatasetFast achieves the highest MRR among baselines. Still, our methods achieve up to 49% higher MRR than DatasetFast. Overall, our methods achieve high performance even with limited training set size.

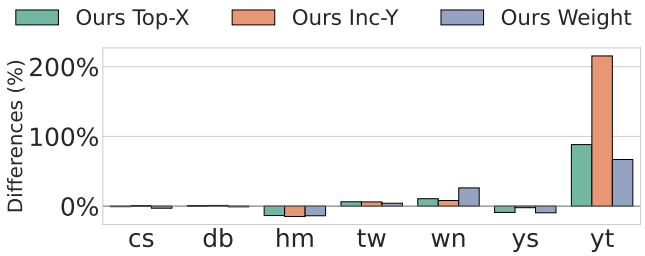


Figure 12: The effect of features.

5.6 Effect of features

To investigate the impact of individual features, we reduce the number of features used for training from 13 to the four features used in RecAlgo. Figure 12 shows the performance gain of versions using all features vs. those using the four features used in RecAlgo, calculated as $(EPS_{all} - EPS_{rec})/EPS_{rec}$, where EPS_{rec} denotes the EPS when using only 4 features and EPS_{all} denotes the EPS when using all 13 features. On db, tw, wn, and yt, performance improves when using all features, while it degrades on cs, hm, and ys. In cs and db, the differences are slight, but in yt, the performance gain is more significant. These results suggest that the features used in RecAlgo are insufficient and it is hard to manually build rule-based models. There is however room for improving performance by feature selection.

Figure 13 shows the Shapley value for each feature, which represents the feature's contribution to model predictions. We find that label ratio, a feature of our own design, has the highest impact, indicating its importance for data and query graphs.

5.7 Training time and epochs

Table 3 shows the average training time and number of epochs. Our methods complete training in at most a few hundred seconds. The training time sometimes takes longer than average, as we use different learning rates chosen from hyper-parameter tuning, leading to high variance. Still, we find that our framework trains machine learning models efficiently.

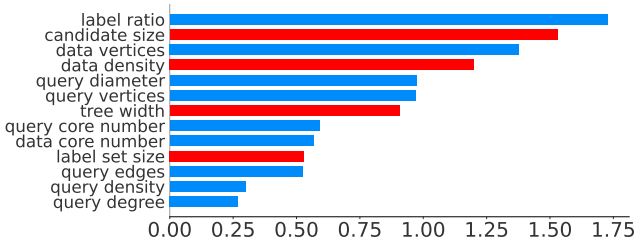


Figure 13: Average of absolute Shapley values. Red and blue bars indicate features used in RecAlgo and added in our method, respectively.

Table 3: Training time and the number of epochs.

Ours	training time (sec)	epochs
Top-X	$48.11_{\pm 24.28}$	$569.00_{\pm 310.64}$
Inc-Y	$44.01_{\pm 17.75}$	$496.40_{\pm 168.57}$
Weight	$95.68_{\pm 119.49}$	$1484.80_{\pm 2126.32}$

6 CONCLUSION AND FUTURE WORK

We introduced the problem of selecting a well-performing subgraph matching algorithm via machine learning and proposed a novel framework for that task. Our experimental study showed that our framework outperforms baselines that select algorithms by rule-based models or use the algorithm with the best average performance. In the future, we aim to study the tailoring of training queries and the selection of features and build foundation models for subgraph matching algorithm selection.

REFERENCES

[1] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. 2019. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 2623–2631.

[2] Yoshua Bengio. 2012. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade: Second edition*. 437–478.

[3] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis Shasha, and Alfredo Ferro. 2013. A subgraph isomorphism algorithm and its application to biochemical data. *BMC bioinformatics* 14 (2013), 1–13.

[4] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis Shasha, and Alfredo Ferro. 2013. A subgraph isomorphism algorithm and its application to biochemical data. *BMC Bioinformatics* 14 (2013), 1–13.

[5] Wenfei Fan. 2012. Graph pattern matching revised for social network analysis. In *Proceedings of the 15th international conference on database theory*. 8–21.

[6] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. 2019. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *Proceedings of the 2019 international conference on management of data*. 1429–1446.

[7] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. 2019. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *Proceedings of the International Conference on Management of Data*. 1429–1446.

[8] Juris Hartmanis. 1982. Computers and intractability: a guide to the theory of np-completeness. *Siam Review* 24, 1 (1982), 90.

[9] Huahai He and Ambuj K Singh. 2008. Graphs-at-a-time: query language and access methods for graph databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 405–418.

[10] Hyunjoon Kim, Yunyoung Choi, Kunsoo Park, Xuemin Lin, Seok-Hee Hong, and Wook-Shin Han. 2021. Versatile Equivalences: Speeding up Subgraph Query Processing and Subgraph Matching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 925–937.

[11] Jinha Kim, Hyungyu Shin, Wook-Shin Han, Sungpack Hong, and Hassan Chafi. 2015. Taming subgraph isomorphism for RDF query processing. *arXiv preprint arXiv:1506.01973* (2015).

[12] Zixun Lan, Ye Ma, Limin Yu, Linglong Yuan, and Fei Ma. 2023. Aednet: Adaptive edge-deleting network for subgraph matching. *Pattern Recognition* 133 (2023), 109033.

[13] Zixun Lan, Limin Yu, Linglong Yuan, Zili Wu, Qiang Niu, and Fei Ma. 2023. Subgmn: The neural subgraph matching network model. In *2023 16th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics*. 1–7.

[14] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. 2012. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *Proceedings of the VLDB Endowment* 6, 2 (2012), 133–144.

[15] Ziming Li, Yuequn Dou, Youhuan Li, Xinhuan Chen, and Chuxu Zhang. 2025. RSM: Reinforced Subgraph Matching Framework with Fine-grained Operation based Search Plan. In *Proceedings of the Eighteenth ACM International Conference on Web Search and Data Mining*. 475–483.

[16] Zhaoyu Lou, Jiaxuan You, Chengtao Wen, Arquimedes Canedo, Jure Leskovec, et al. 2020. Neural subgraph matching. *arXiv preprint arXiv:2007.03092* (2020).

[17] Michael Ogunsanya, Joan Isichei, and Salil Desai. 2023. Grid search hyperparameter tuning in additive manufacturing processes. *Manufacturing Letters* 35 (2023), 1031–1042.

[18] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2009. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems* 34, 3 (2009), 1–45.

[19] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time constrained cycle detection in large dynamic graphs. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1876–1888.

[20] Neil Robertson and Paul D. Seymour. 1986. Graph minors. II. Algorithmic aspects of tree-width. *Journal of algorithms* 7, 3 (1986), 309–322.

[21] Reuven Rubinstein. 1999. The cross-entropy method for combinatorial and continuous optimization. *Methodology and computing in applied probability* 1 (1999), 127–190.

[22] Stephen B Seidman. 1983. Network structure and minimum degree. *Social networks* 5 (1983), 269–287.

[23] Konstantinos Skitsas, Davide Mottin, and Panagiotis Karras. 2025. PILOS: Scalable Large-Subgraph Matching by Online Spectral Filtering. In *2025 IEEE 41st International Conference on Data Engineering*. 1180–1193.

[24] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research* 15, 1 (2014), 1929–1958.

[25] Shixuan Sun and Qiong Luo. 2020. In-Memory Subgraph Matching: An In-depth Study. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1083–1098.

[26] Shixuan Sun, Xibo Sun, Yulin Che, Qiong Luo, and Bingsheng He. 2020. RapidMatch: a holistic approach to subgraph query processing. *Proceedings of the VLDB Endowment* 14, 2 (2020), 176–188.

[27] Julian R Ullmann. 1976. An algorithm for subgraph isomorphism. *J. ACM* 23, 1 (1976), 31–42.

[28] Todd L Veldhuizen. 2012. Leapfrog triejoin: a worst-case optimal join algorithm. *arXiv preprint arXiv:1210.0481* (2012).

[29] Hanchen Wang, Ying Zhang, Lu Qin, Wei Wang, Wenjie Zhang, and Xuemin Lin. 2022. Reinforcement learning based query vertex ordering model for subgraph matching. In *2022 IEEE 38th International Conference on Data Engineering*. 245–258.

[30] Rongjian Yang, Zhijie Zhang, Weiguo Zheng, and Jeffrey Xu Yu. 2023. Fast Continuous Subgraph Matching over Streaming Graphs via Backtracking Reduction. *Proceedings of the ACM on Management of Data* 1, 1, Article 15 (2023), 26 pages.

[31] Zhijie Zhang, Yujie Lu, Weiguo Zheng, and Xuemin Lin. 2024. A Comprehensive Survey and Experimental Study of Subgraph Matching: Trends, Unbiasedness, and Interaction. *Proceedings of the ACM on Management of Data* (2024), 1–29.

[32] Gaoping Zhu, Xuemin Lin, Ke Zhu, Wenjie Zhang, and Jeffrey Xu Yu. 2012. TreeSpan: efficiently computing similarity all-matching. In *Proceedings of the 2012 ACM SIGMOD international conference on management of data*. 529–540.

Table 7: Inference cost

Dataset	Cost ratio
cs	0.773
db	0.962
hm	0.928
tw	0.979
wn	0.979
ys	0.574
yt	0.975

Table 4: Rules of RecAlgo.

filtering	DPiso	default
	LDF / NLF	high selectivity
	VEQ	low selectivity
ordering	RI / RM	default
	GQL	sparse dataset & small candidate size
enumeration	KSS	small tree width
	VEQ	default
	KSS	large candidate size

Table 5: EPS Performance Across Datasets

EPS	cs	db	hm	tw	wn	ys	yt
Oracle	1.14e+8	1.08e+8	7.55e+7	6.99e+5	3.59e+6	1.05e+9	1.67e+7
DatasetFast	**1.05e+8**	6.61e+7	3.58e+5	3.51e+5	3.04e+5	2.56e+7	4.32e+5
AllFast	3.61e+7	4.26e+7	4.13e+7	1.47e+5	1.26e+6	**3.31e+8**	6.49e+6
RecAlgo	8.35e+6	1.38e+7	3.81e+7	2.26e+5	4.34e+5	3.22e+6	3.90e+6
Ours Top-X	6.28e+7	**7.09e+7**	3.91e+7	**5.63e+5**	1.77e+6	1.31e+8	2.88e+6
Ours Inc-Y	6.32e+7	6.97e+7	3.79e+7	5.62e+5	1.61e+6	1.31e+8	4.10e+6
Ours Weight	6.22e+7	6.86e+7	**4.62e+7**	5.48e+5	**2.22e+6**	1.33e+8	**7.76e+6**

Table 6: MRR Performance Across Datasets

MRR	cs	db	hm	tw	wn	ys	yt
Oracle	1.0	1.0	1.0	1.0	1.0	1.0	1.0
DatasetFast	0.17	0.24	0.12	0.20	0.18	0.22	0.15
AllFast	0.15	0.18	0.11	0.06	0.08	0.18	0.10
RecAlgo	0.10	0.10	0.12	0.10	0.08	0.09	0.14
Top-X	**0.29**	**0.37**	**0.22**	**0.38**	**0.32**	**0.36**	**0.27**
Inc-Y	**0.29**	0.36	0.21	**0.38**	**0.32**	0.35	0.26
Weighted	0.27	0.34	0.20	0.34	0.31	0.35	0.23

A APPENDIX

A.1 Rules of RecAlgo

The rules of RecAlgo are listed in Table 4. The threshold for each feature, such as high or low values, is defined as the top 10% or bottom 10% of the entire dataset. Also, in the table, the conditions listed higher have higher priority.

A.2 An average EPS for each rank in each query set Q_I and Q_S

Figure 14 show the results decomposing the results in Figure 2 into Q_I and Q_S. In tw and wn, it is observed that Q_I shows a smooth change, whereas Q_S shows a more stepwise change.

A.3 The detail of the results

Table 5 and 6 show the row data in Figure 5.

A.4 Impact of training data size on EPS across datasets

In section 5.7, we show the impact of training data size on MRR. Here, we evaluate that with EPS and results are shown in Figure 15. When evaluating with EPS, reducing the training data size leads to a significant improvement in EPS on hm, wn, ys, and yt. This is likely because, as shown in Table 8, the algorithms selected by DatasetFast on these datasets exhibit an EPS difference lower than one order of magnitude compared to the algorithms that achieve the highest EPS on average.

A.5 An Average EPS for each Algorithm

An average EPS for each algorithm are listed in Table 8.

A.6 Inference Cost of Our Methods

Table 7 shows the cost of the inference, calculated by $EPS_{w/Cost}/EPS_{w/oCost}$, $EPS_{w/Cost}$ denotes the EPS of our method and $EPS_{w/oCost}$ denotes the EPS of our method ignoring the cost of inference and featurizer.

A.7 EPS on Each Query Size for Each Query Set

Figure 16 shows the results from Figure 9 (a) and (b), summarized by each query size.

A.8 Shapley Value Analyses

Figure 17 illustrates the contribution of each feature to our method's predictions. Here, we use Top-X as our method.

Here, we show the impact that each feature has on each algorithm's prediction, as well as the correlation between those predictions and the feature values.

Table 8: Average EPS for Each Algorithm, results of DatasetFast is indicated in bold and those of AllFast is emphasized with underline

Algorithm	cs	db	hm	tw	wn	ys	yt
DPiso-GQL-EXPLORE	1.32e+05	1.41e+05	2.47e+05	1.93e+05	2.17e+05	2.05e+05	2.83e+05
DPiso-GQL-KSS	1.33e+06	1.84e+05	1.73e+06	2.83e+05	**2.92e+05**	**2.67e+07**	4.42e+05
DPiso-GQL-LFTJ	1.52e+05	1.85e+05	**3.65e+05**	3.04e+05	3.15e+05	2.33e+05	4.27e+05
DPiso-GQL-VEQ	1.48e+07	4.15e+07	2.43e+07	5.87e+04	1.15e+06	1.16e+09	8.29e+06
DPiso-RI-EXPLORE	1.38e+05	1.45e+05	2.14e+05	2.10e+05	2.39e+05	1.79e+05	3.15e+05
DPiso-RI-KSS	1.24e+06	1.85e+05	1.45e+06	2.92e+05	2.75e+05	1.65e+07	4.59e+05
DPiso-RI-LFTJ	1.42e+05	1.88e+05	2.61e+05	3.34e+05	3.20e+05	1.92e+05	4.34e+05
DPiso-RI-VEQ	1.39e+07	4.11e+07	2.79e+07	5.87e+04	1.39e+06	8.40e+08	8.29e+06
DPiso-RM-EXPLORE	1.53e+05	1.44e+05	2.20e+05	2.21e+05	2.40e+05	2.03e+05	3.10e+05
DPiso-RM-KSS	1.09e+06	1.89e+05	1.48e+06	3.09e+05	**2.92e+05**	2.17e+07	4.52e+05
DPiso-RM-LFTJ	1.65e+05	1.86e+05	2.79e+05	3.53e+05	**3.39e+05**	2.25e+05	4.40e+05
DPiso-RM-VEQ	1.46e+07	4.07e+07	2.74e+07	5.85e+04	1.28e+06	9.38e+08	7.10e+06
LDF-GQL-EXPLORE	9.39e+04	1.14e+05	1.64e+05	1.58e+05	2.05e+05	1.52e+05	2.39e+05
LDF-GQL-KSS	1.40e+05	1.15e+05	4.56e+05	2.29e+05	2.26e+05	2.16e+06	3.40e+05
LDF-GQL-LFTJ	1.19e+05	1.76e+05	2.00e+05	2.51e+05	2.78e+05	1.73e+05	3.86e+05
LDF-GQL-VEQ	5.88e+04	1.88e+06	7.06e+06	2.47e+04	8.35e+04	2.07e+06	3.04e+06
LDF-RI-EXPLORE	1.03e+05	1.18e+05	1.36e+05	1.81e+05	2.10e+05	1.01e+05	2.65e+05
LDF-RI-KSS	1.49e+05	1.08e+05	3.86e+05	2.46e+05	2.30e+05	8.46e+05	3.51e+05
LDF-RI-LFTJ	1.19e+05	1.76e+05	1.73e+05	3.27e+05	2.99e+05	1.55e+05	4.12e+05
LDF-RI-VEQ	5.79e+04	1.67e+06	6.81e+06	2.46e+04	8.50e+04	2.58e+06	2.93e+06
LDF-RM-EXPLORE	1.01e+05	1.20e+05	1.53e+05	1.91e+05	2.21e+05	1.67e+05	2.66e+05
LDF-RM-KSS	1.49e+05	1.19e+05	4.08e+05	2.61e+05	2.28e+05	1.12e+06	3.57e+05
LDF-RM-LFTJ	1.18e+05	1.76e+05	1.72e+05	3.40e+05	3.06e+05	1.79e+05	4.20e+05
LDF-RM-VEQ	5.74e+04	1.75e+06	6.09e+06	2.46e+04	8.46e+04	2.19e+06	2.79e+06
NLF-GQL-EXPLORE	1.40e+05	1.77e+05	2.37e+05	2.26e+05	2.34e+05	1.67e+05	2.94e+05
NLF-GQL-KSS	5.77e+05	2.06e+05	2.17e+06	3.29e+05	2.77e+05	1.40e+07	4.55e+05
NLF-GQL-LFTJ	1.48e+05	1.84e+05	**3.38e+05**	3.11e+05	2.93e+05	2.09e+05	4.22e+05
NLF-GQL-VEQ	4.51e+05	2.43e+07	1.75e+07	6.85e+04	6.73e+05	1.97e+08	8.35e+06
NLF-RI-EXPLORE	1.44e+05	1.79e+05	2.10e+05	2.52e+05	2.66e+05	1.73e+05	3.28e+05
NLF-RI-KSS	5.46e+05	1.94e+05	1.63e+06	3.41e+05	2.75e+05	1.19e+07	4.60e+05
NLF-RI-LFTJ	1.38e+05	1.85e+05	2.38e+05	3.39e+05	3.24e+05	1.81e+05	4.26e+05
NLF-RI-VEQ	3.91e+05	2.41e+07	1.67e+07	6.85e+04	6.11e+05	1.98e+08	7.38e+06
NLF-RM-EXPLORE	1.44e+05	1.83e+05	2.08e+05	2.58e+05	2.58e+05	1.81e+05	3.23e+05
NLF-RM-KSS	5.87e+05	2.08e+05	2.03e+06	**3.61e+05**	2.73e+05	1.24e+07	4.67e+05
NLF-RM-LFTJ	1.45e+05	1.83e+05	2.48e+05	**3.60e+05**	3.30e+05	2.10e+05	**4.42e+05**
NLF-RM-VEQ	3.20e+05	2.33e+07	1.77e+07	6.80e+04	6.75e+05	2.22e+08	7.56e+06
PILOS-GQL-EXPLORE	1.33e+05	1.00e+05	1.89e+05	1.94e+05	1.80e+05	1.52e+05	2.42e+05
PILOS-GQL-KSS	7.79e+05	2.25e+05	1.29e+06	3.28e+05	2.86e+05	7.80e+06	4.62e+05
PILOS-GQL-LFTJ	9.04e+04	1.35e+05	2.51e+05	2.46e+05	1.61e+05	1.17e+05	2.26e+05
PILOS-GQL-VEQ	1.09e+07	4.91e+07	5.27e+06	4.60e+04	1.89e+06	2.36e+08	7.20e+06
PILOS-RI-EXPLORE	1.17e+05	1.06e+05	1.71e+05	1.96e+05	1.88e+05	1.29e+05	2.57e+05
PILOS-RI-KSS	6.41e+05	2.27e+05	1.06e+06	3.36e+05	2.78e+05	5.98e+06	4.76e+05
PILOS-RI-LFTJ	8.07e+04	1.32e+05	2.17e+05	2.45e+05	1.57e+05	1.01e+05	2.25e+05
PILOS-RI-VEQ	1.11e+07	**4.92e+07**	5.96e+06	4.58e+04	1.79e+06	2.47e+08	6.99e+06
PILOS-RM-EXPLORE	1.35e+05	1.01e+05	1.72e+05	2.02e+05	1.87e+05	1.46e+05	2.54e+05
PILOS-RM-KSS	7.04e+05	2.25e+05	1.17e+06	3.49e+05	2.92e+05	7.36e+06	4.69e+05
PILOS-RM-LFTJ	9.13e+04	1.31e+05	2.24e+05	2.54e+05	1.62e+05	1.13e+05	2.27e+05
PILOS-RM-VEQ	1.05e+07	4.88e+07	5.42e+06	4.60e+04	1.85e+06	2.45e+08	7.40e+06
VEQ-GQL-EXPLORE	1.43e+05	1.63e+05	2.69e+05	1.91e+05	2.00e+05	2.10e+05	2.81e+05
VEQ-GQL-KSS	7.41e+05	2.30e+05	1.28e+06	2.75e+05	2.66e+05	1.29e+07	4.36e+05
VEQ-GQL-LFTJ	1.47e+05	1.76e+05	3.70e+05	2.40e+05	2.47e+05	2.26e+05	3.87e+05
VEQ-GQL-VEQ	**1.33e+08**	**6.46e+07**	3.78e+07	6.17e+04	2.14e+06	9.83e+08	9.44e+06
VEQ-RI-EXPLORE	1.42e+05	1.72e+05	2.40e+05	2.03e+05	2.16e+05	1.79e+05	3.09e+05
VEQ-RI-KSS	7.28e+05	2.36e+05	1.26e+06	2.79e+05	2.51e+05	9.79e+06	4.54e+05
VEQ-RI-LFTJ	1.29e+05	1.73e+05	2.83e+05	2.45e+05	2.44e+05	1.77e+05	3.84e+05
VEQ-RI-VEQ	1.16e+08	**6.58e+07**	3.57e+07	6.16e+04	2.27e+06	8.23e+08	9.24e+06
VEQ-RM-EXPLORE	1.54e+05	1.66e+05	2.43e+05	2.08e+05	2.13e+05	2.03e+05	3.01e+05
VEQ-RM-KSS	7.60e+05	2.35e+05	1.36e+06	2.90e+05	2.72e+05	1.19e+07	4.43e+05
VEQ-RM-LFTJ	1.53e+05	1.71e+05	3.01e+05	2.56e+05	2.57e+05	2.09e+05	3.91e+05
VEQ-RM-VEQ	1.21e+08	6.47e+07	3.61e+07	6.09e+04	2.08e+06	9.33e+08	9.66e+06

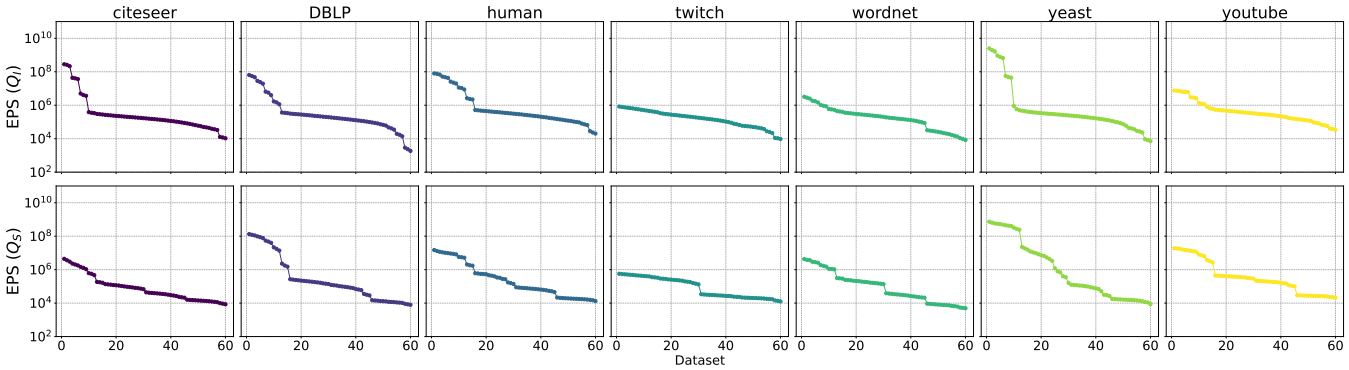


Figure 14: A mean EPS for each rank with Q_I(above) and Q_S(below).

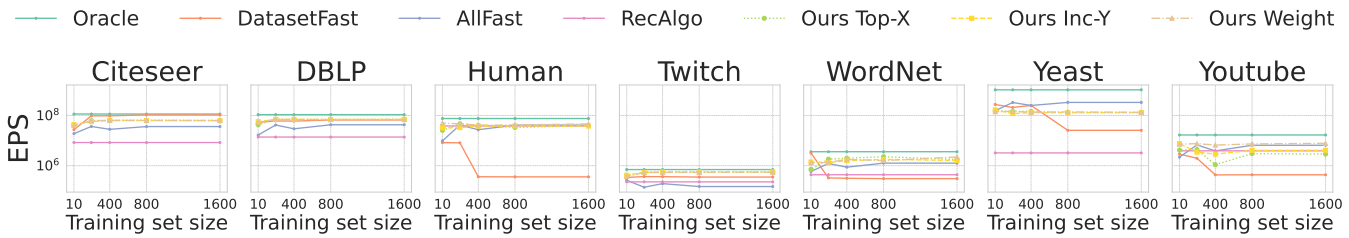


Figure 15: Effect of training data size on EPS across datasets.

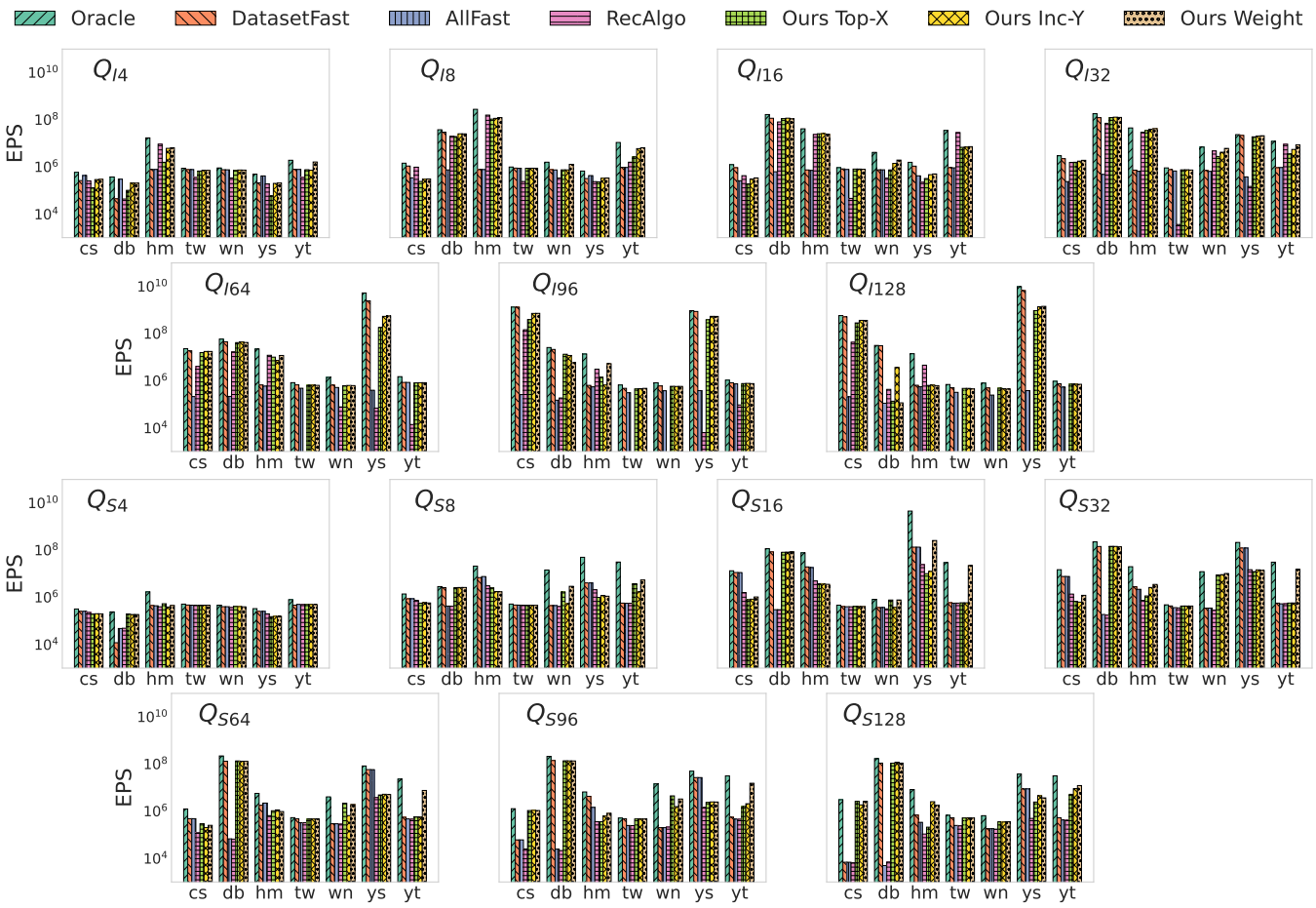
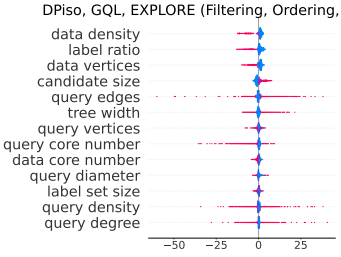


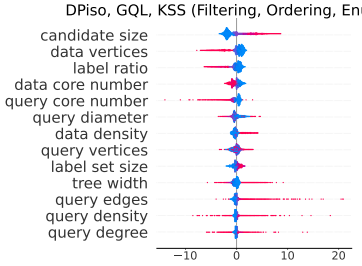
Figure 16: EPS on each query size.

Feature value

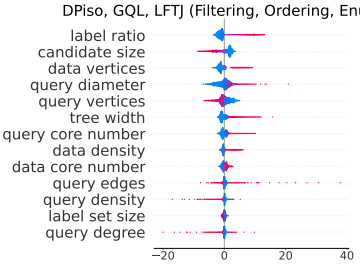
DPiso, GQL, EXPLORE (Filtering, Ordering, Enumeration)



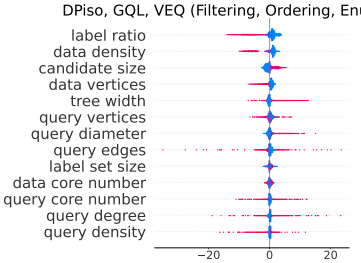
DPiso, GQL, KSS (Filtering, Ordering, Enumeration)



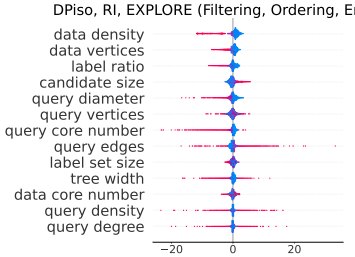
DPiso, GQL, LFTJ (Filtering, Ordering, Enumeration)



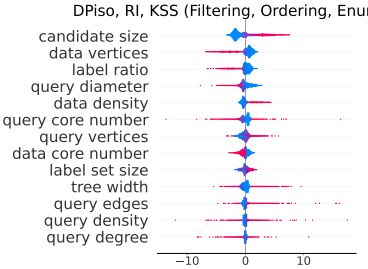
DPiso, GQL, VEQ (Filtering, Ordering, Enumeration)



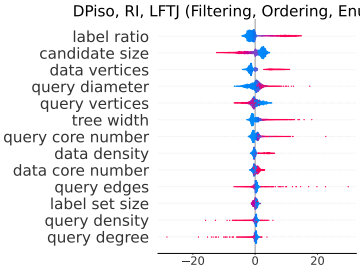
DPiso, RI, EXPLORE (Filtering, Ordering, Enumeration)



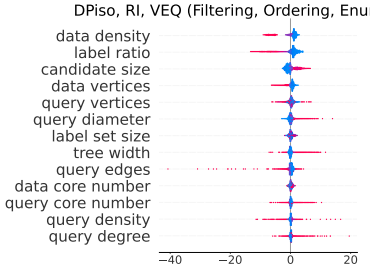
DPiso, RI, KSS (Filtering, Ordering, Enumeration)



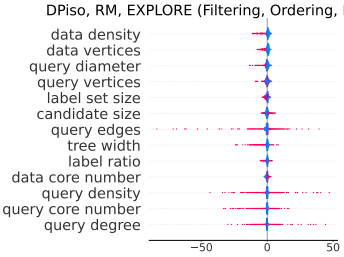
DPiso, RI, LFTJ (Filtering, Ordering, Enumeration)



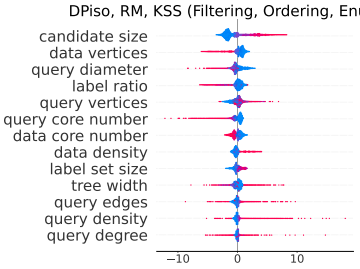
DPiso, RI, VEQ (Filtering, Ordering, Enumeration)



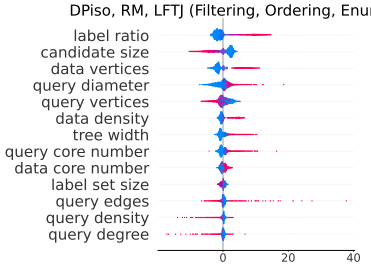
DPiso, RM, EXPLORE (Filtering, Ordering, Enumeration)



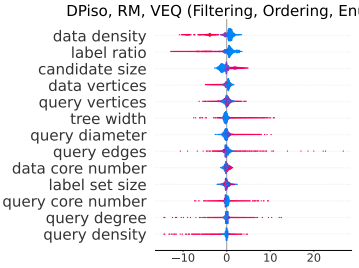
DPiso, RM, KSS (Filtering, Ordering, Enumeration)



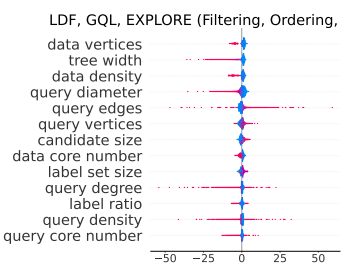
DPiso, RM, LFTJ (Filtering, Ordering, Enumeration)



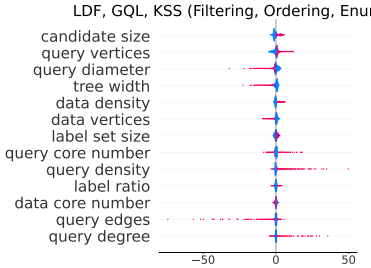
DPiso, RM, VEQ (Filtering, Ordering, Enumeration)



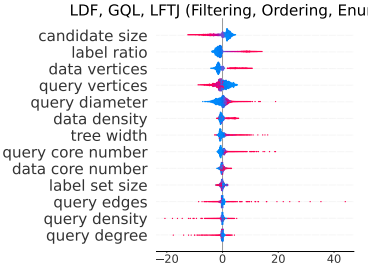
LDF, GQL, EXPLORE (Filtering, Ordering, Enumeration)

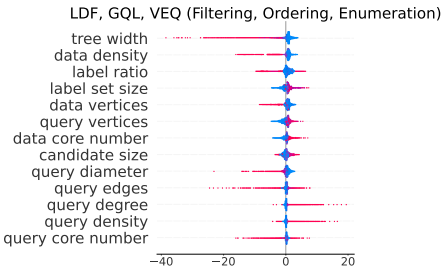


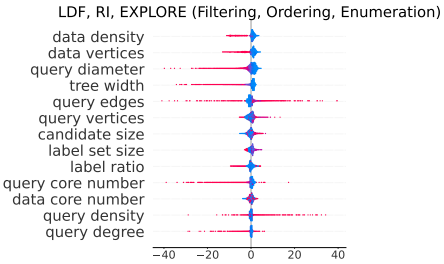
LDF, GQL, KSS (Filtering, Ordering, Enumeration)

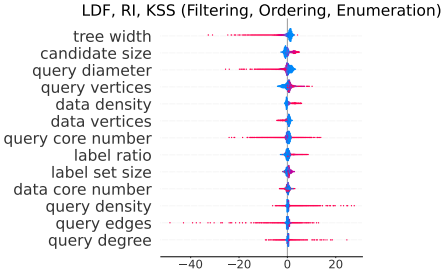


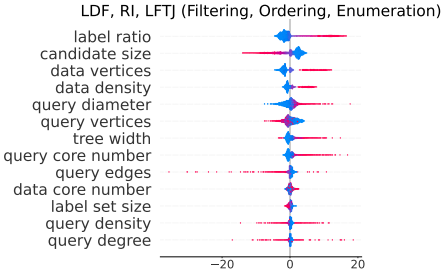
LDF, GQL, LFTJ (Filtering, Ordering, Enumeration)

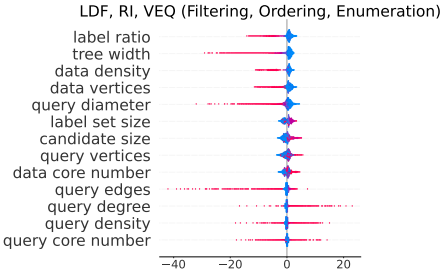


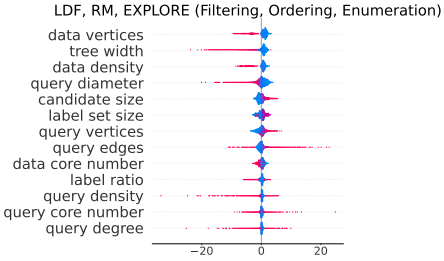

LDF, GQL, VEQ (Filtering, Ordering, Enumeration)
tree width
data density
label ratio
label set size
data vertices
query vertices
data core number
candidate size
query diameter
query edges
query degree
query density
query core number
−40
−20
0
20

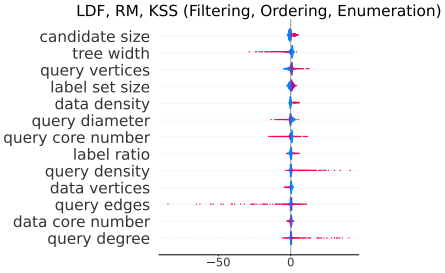

LDF, RI, EXPLORE (Filtering, Ordering, Enumeration)
data density
data vertices
query diameter
tree width
query edges
query vertices
candidate size
label set size
label ratio
query core number
data core number
query density
query degree
−40
−20
0
20
40

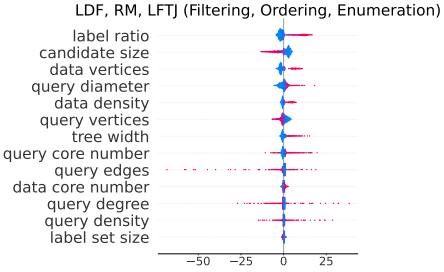

LDF, RI, KSS (Filtering, Ordering, Enumeration)
tree width
candidate size
query diameter
query vertices
data density
data vertices
query core number
label ratio
label set size
data core number
query density
query edges
query degree
−40
−20
0
20

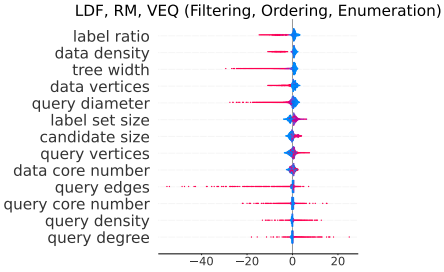

LDF, RI, LFTJ (Filtering, Ordering, Enumeration)
label ratio
candidate size
data vertices
data density
query diameter
query vertices
tree width
query core number
query edges
data core number
label set size
query density
query degree
−20
0
20

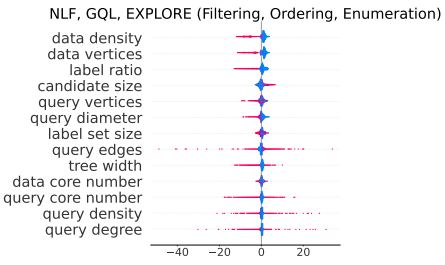

LDF, RI, VEQ (Filtering, Ordering, Enumeration)
label ratio
tree width
data density
data vertices
query diameter
label set size
candidate size
query vertices
data core number
query edges
query degree
query density
query core number
−40
−20
0
20

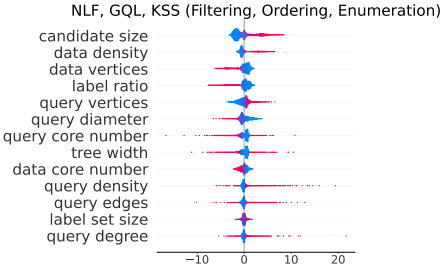

LDF, RM, EXPLORE (Filtering, Ordering, Enumeration)
data vertices
tree width
data density
query diameter
candidate size
label set size
query vertices
query edges
data core number
label ratio
query density
query core number
query degree
−20
0
20

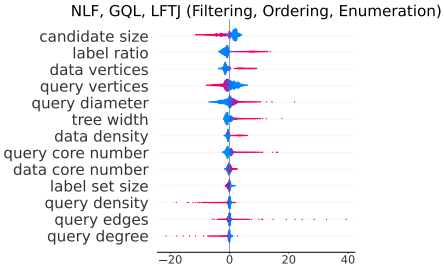

LDF, RM, KSS (Filtering, Ordering, Enumeration)
candidate size
tree width
query vertices
label set size
data density
query diameter
query core number
label ratio
query density
data vertices
query edges
data core number
query degree
−50
0

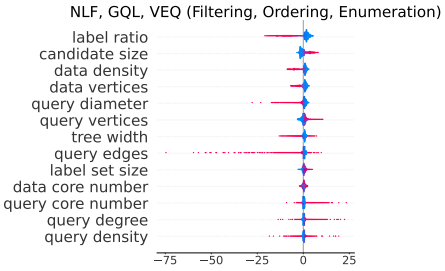

LDF, RM, LFTJ (Filtering, Ordering, Enumeration)
label ratio
candidate size
data vertices
query diameter
data density
query vertices
tree width
query core number
query edges
data core number
query degree
query density
label set size
−50
−25
0
25

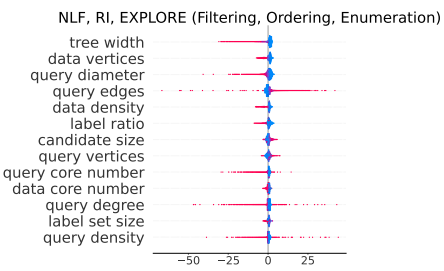

LDF, RM, VEQ (Filtering, Ordering, Enumeration)
label ratio
data density
tree width
data vertices
query diameter
label set size
candidate size
query vertices
data core number
query edges
query core number
query density
query degree
−40
−20
0
20

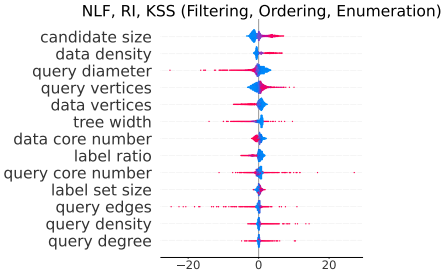

NLF, GQL, EXPLORE (Filtering, Ordering, Enumeration)
data density
data vertices
label ratio
candidate size
query vertices
query diameter
label set size
query edges
tree width
data core number
query core number
query density
query degree
−40
−20
0
20

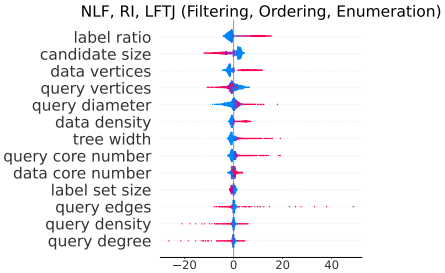

NLF, GQL, KSS (Filtering, Ordering, Enumeration)
candidate size
data density
data vertices
label ratio
query vertices
query diameter
query core number
tree width
data core number
query density
query edges
label set size
query degree
−10
0
10
20

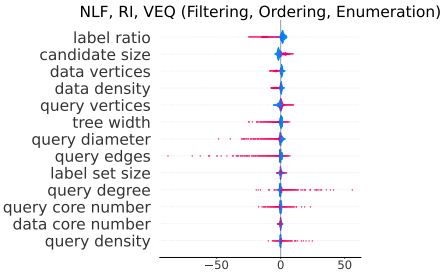

NLF, GQL, LFTJ (Filtering, Ordering, Enumeration)
candidate size
label ratio
data vertices
query vertices
query diameter
tree width
data density
query core number
data core number
label set size
query density
query edges
query degree
−20
0
20
40

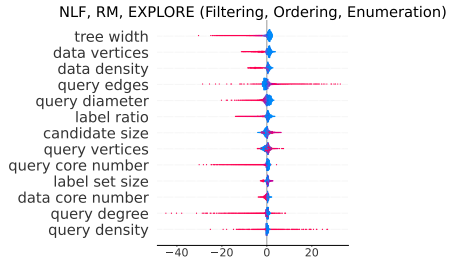

NLF, GQL, VEQ (Filtering, Ordering, Enumeration)
label ratio
candidate size
data density
data vertices
query diameter
query vertices
tree width
query edges
label set size
data core number
query core number
query degree
query density
−75
−50
−25
0
25

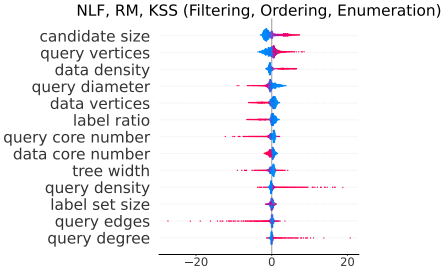

NLF, RI, EXPLORE (Filtering, Ordering, Enumeration)
tree width
data vertices
query diameter
query edges
data density
label ratio
candidate size
query vertices
query core number
data core number
query degree
label set size
query density
−50
−25
0
25

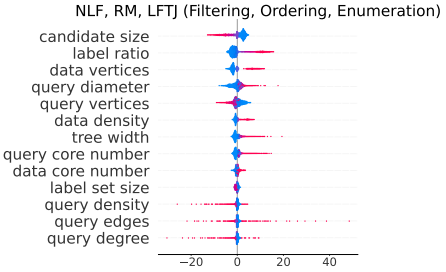

NLF, RI, KSS (Filtering, Ordering, Enumeration)
candidate size
data density
query diameter
query vertices
data vertices
tree width
data core number
label ratio
query core number
label set size
query edges
query density
query degree
−20
0
20

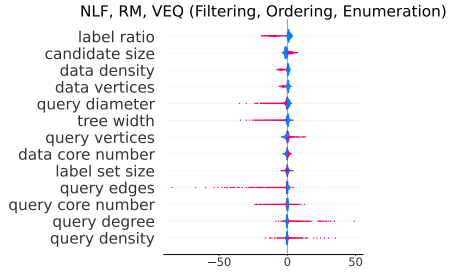

NLF, RI, LFTJ (Filtering, Ordering, Enumeration)
label ratio
candidate size
data vertices
query vertices
query diameter
data density
tree width
query core number
data core number
label set size
query edges
query density
query degree
−20
0
20
40

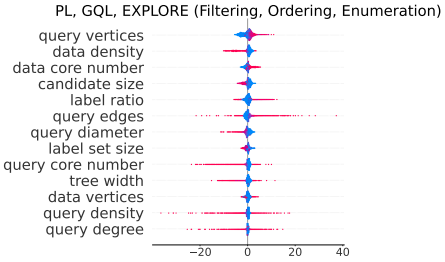

NLF, RI, VEQ (Filtering, Ordering, Enumeration)
label ratio
candidate size
data vertices
data density
query vertices
tree width
query diameter
query edges
label set size
query degree
query core number
data core number
query density
−50
0
50

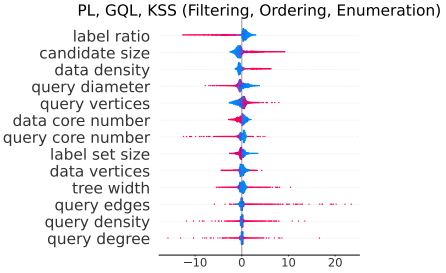

NLF, RM, EXPLORE (Filtering, Ordering, Enumeration)
tree width
data vertices
data density
query edges
query diameter
label ratio
candidate size
query vertices
query core number
label set size
data core number
query degree
query density
−40
−20
0
20

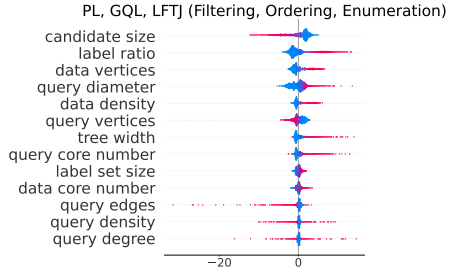

NLF, RM, KSS (Filtering, Ordering, Enumeration)
candidate size
query vertices
data density
query diameter
data vertices
label ratio
query core number
data core number
tree width
query density
label set size
query edges
query degree
−20
0
20

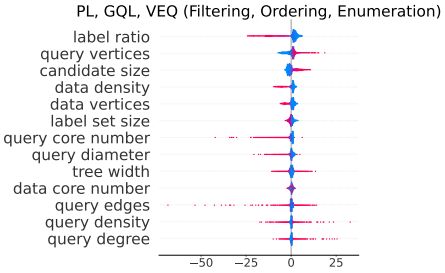

NLF, RM, LFTJ (Filtering, Ordering, Enumeration)
candidate size
label ratio
data vertices
query diameter
query vertices
data density
tree width
query core number
data core number
label set size
query density
query edges
query degree
−20
0
20
40

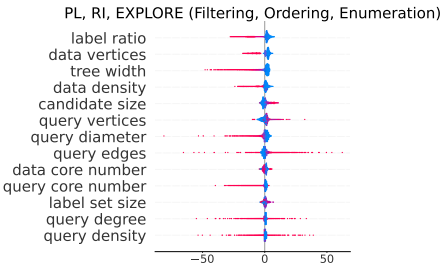

NLF, RM, VEQ (Filtering, Ordering, Enumeration)
label ratio
candidate size
data density
data vertices
query diameter
tree width
query vertices
data core number
label set size
query edges
query core number
query degree
query density
−50
0
50

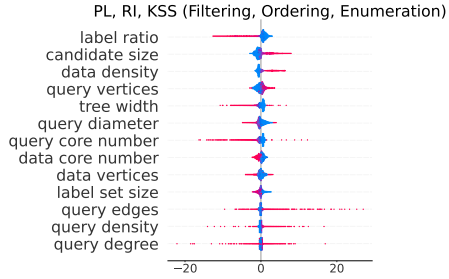

PL, GQL, EXPLORE (Filtering, Ordering, Enumeration)
query vertices
data density
data core number
candidate size
label ratio
query edges
query diameter
label set size
query core number
tree width
data vertices
query density
query degree
−20
0
20
40

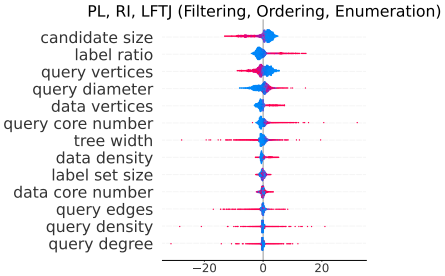

PL, GQL, KSS (Filtering, Ordering, Enumeration)
label ratio
candidate size
data density
query diameter
query vertices
data core number
query core number
label set size
data vertices
tree width
query edges
query density
query degree
−10
0
10
20

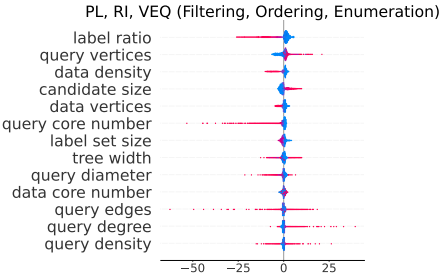

PL, GQL, LFTJ (Filtering, Ordering, Enumeration)
candidate size
label ratio
data vertices
query diameter
data density
query vertices
tree width
query core number
label set size
data core number
query edges
query density
query degree
−20
0

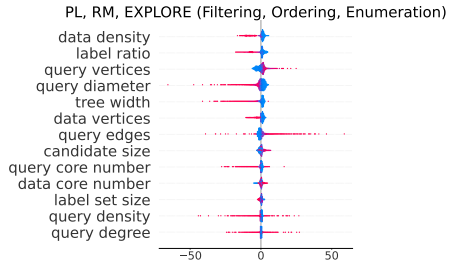

PL, GQL, VEQ (Filtering, Ordering, Enumeration)
label ratio
query vertices
candidate size
data density
data vertices
label set size
query core number
query diameter
tree width
data core number
query edges
query density
query degree
−50
−25
0
25


PL, RI, EXPLORE (Filtering, Ordering, Enumeration)
label ratio
data vertices
tree width
data density
candidate size
query vertices
query diameter
query edges
data core number
query core number
label set size
query degree
query density
−50
0
50


PL, RI, KSS (Filtering, Ordering, Enumeration)
label ratio
candidate size
data density
query vertices
tree width
query diameter
query core number
data core number
data vertices
label set size
query edges
query density
query degree
−20
0
20


PL, RI, LFTJ (Filtering, Ordering, Enumeration)
candidate size
label ratio
query vertices
query diameter
data vertices
query core number
tree width
data density
label set size
data core number
query edges
query density
query degree
−20
0
20


PL, RI, VEQ (Filtering, Ordering, Enumeration)
label ratio
query vertices
data density
candidate size
data vertices
query core number
label set size
tree width
query diameter
data core number
query edges
query degree
query density
−50
−25
0
25


PL, RM, EXPLORE (Filtering, Ordering, Enumeration)
data density
label ratio
query vertices
query diameter
tree width
data vertices
query edges
candidate size
query core number
data core number
label set size
query density
query degree
−50
0
50

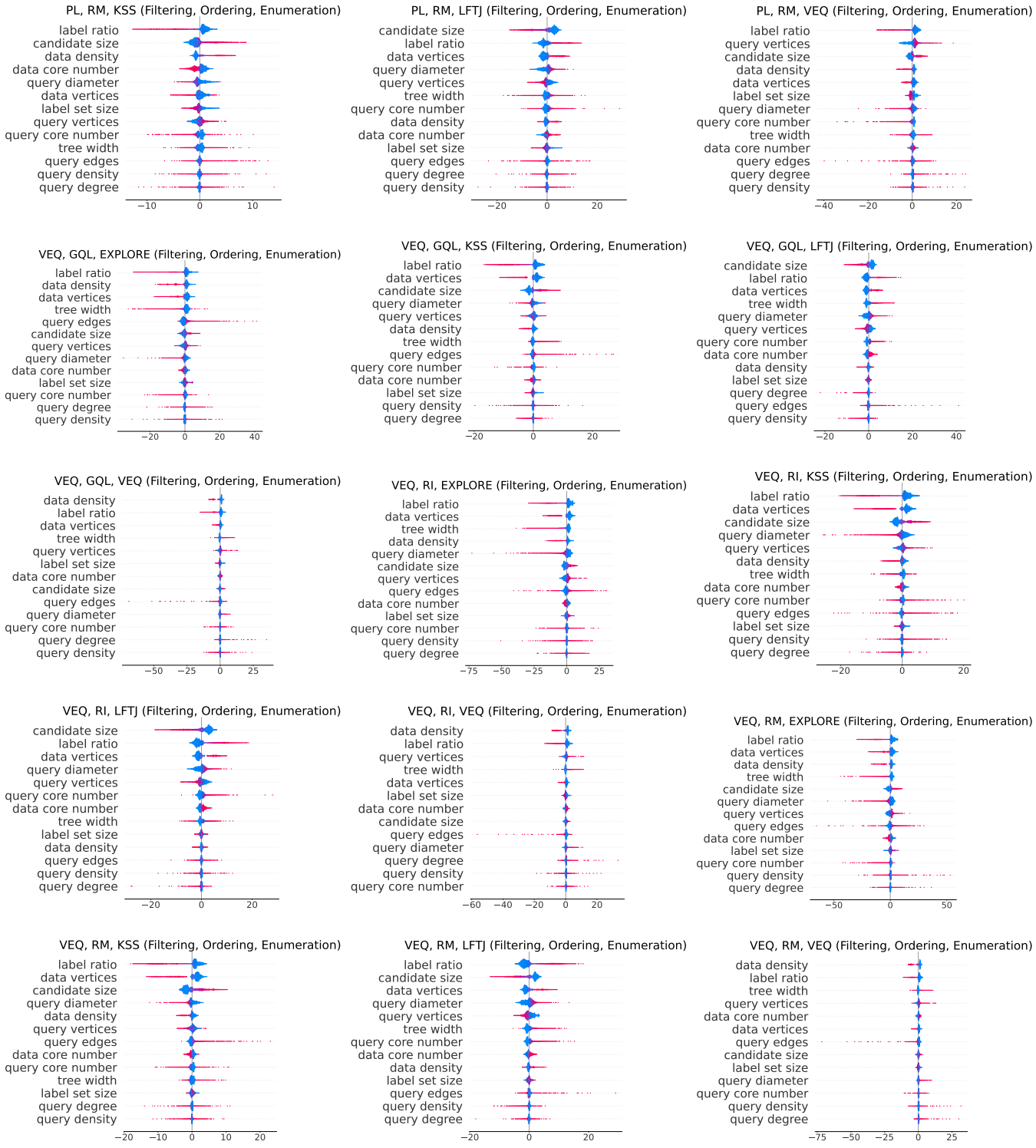


Figure 17: Shapley value analysis for each class.