2016

Learning To Scale Up Search-Driven Data Integration

Zhepeng Yan
University of Pennsylvania, zhepeng@cis.upenn.edu

Learning To Scale Up Search-Driven Data Integration

Abstract
A recent movement to tackle the long-standing data integration problem is a compositional and iterative approach, termed "pay-as-you-go" data integration. Under this model, the objective is to immediately support queries over "partly integrated" data, and to enable the user community to drive integration of the data that relate to their actual information needs. Over time, data will be gradually integrated.

While the pay-as-you-go vision has been well-articulated for some time, only recently have we begun to understand how it can be manifested into a system implementation. One branch of this effort has focused on enabling queries through keyword search-driven data integration, in which users pose queries over partly integrated data encoded as a graph, receive ranked answers generated from data and metadata that is linked at query-time, and provide feedback on those answers. From this user feedback, the system learns to repair bad schema matches or record links.

Many real world issues of uncertainty and diversity in search-driven integration remain open. Such tasks in search-driven integration require a combination of human guidance and machine learning. The challenge is how to make maximal use of limited human input. This thesis develops three methods to scale up search-driven integration, through learning from expert feedback: (1) active learning techniques to repair links from small amounts of user feedback; (2) collaborative learning techniques to combine users' conflicting feedback; and (3) debugging techniques to identify where data experts could best improve integration

quality. We implement these methods within the Q System, a prototype of search-driven integration, and validate their effectiveness over real-world datasets.

Degree Type
Dissertation

Degree Name
Doctor of Philosophy (PhD)

Graduate Group
Computer and Information Science

First Advisor
Zachary G. Ives

Subject Categories
Computer Sciences

LEARNING TO SCALE UP SEARCH-DRIVEN DATA INTEGRATION

Zhepeng Yan

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2016

To my father Guoqiang Yan, mother Jingjuan He, and wife Tianjiao Zhang.

Acknowledgements

I am greatly indebted to my advisor Zachary Ives, who is always available to listen to my (sometimes naive) ideas, offer me insightful comments, teach me how to be a mature researcher, and give me encouragement, despite his very busy schedule. It has been an absolute privilege and an incredible experience working with and learning from Zachary Ives.

I am extremely grateful to Val Tannen, who has taught me theories of data exchange and database provenance. He has been a great mentor during my Ph.D career. I also wish to thank other members in the Penn Database Group, especially Susan Davidsan and Boon Thau Loo, for valuable discussions in regular group seminars. I am also thankful to Cong Yu and Boulos Harb for a fruitful internship at Google.

From the CIS Department, I am grateful to Sudipto Guha and Lyle Ungar for serving on my WPE-II and dissertation committee and suggesting numerous improvements to my thesis. I have had a great amount of fun playing chess with Sanjeev Khanna, who has also helped me start my Ph.D journey.

I wish to thank my friends at Penn, especially Mingchen Zhao, Yifei Yuan, Chen Chen, and Yang Li, for making my graduate school life memorable. I am also grateful to You Wu for his friendship.

And above all, I thank my parents for their unconditional support and love, without which this would have been impossible.

Last but not least, I thank my wife Tianjiao Zhang for her love and sacrifice. She has been working in California since 2011 and long-distance relationship is not easy for us. I look forward to starting a new chapter of my life with her.

This work was funded in part by the following grants: NSF IIS-0513778, IIS-1217798, ACI-1547360, NIH 1U01EB02954, and a gift from Google.

ABSTRACT

LEARNING TO SCALE UP SEARCH-DRIVEN DATA INTEGRATION

Zhepeng Yan

Zachary G. Ives

A recent movement to tackle the long-standing data integration problem is a compositional and iterative approach, termed "pay-as-you-go" data integration. Under this model, the objective is to immediately support queries over "partly integrated" data, and to enable the user community to drive integration of the data that relate to their actual information needs. Over time, data will be gradually integrated.

While the pay-as-you-go vision has been well-articulated for some time, only recently have we begun to understand how it can be manifested into a system implementation. One branch of this effort has focused on enabling queries through keyword search-driven data integration, in which users pose queries over partly integrated data encoded as a graph, receive ranked answers generated from data and metadata that is linked at query-time, and provide feedback on those answers. From this user feedback, the system learns to repair bad schema matches or record links.

Many real world issues of uncertainty and diversity in search-driven integration remain open. Such tasks in search-driven integration require a combination of human guidance and machine learning. The challenge is how to make maximal use of limited human input. This thesis develops three methods to scale up search-driven integration, through learning from expert feedback: (1) active learning techniques to repair links from small amounts of user feedback; (2) collaborative learning techniques to combine users' conflicting feedback; and (3) debugging techniques to identify where data experts could best improve integration quality. We implement these methods within the **Q** System, a prototype of search-driven integration, and validate their effectiveness over real-world datasets.

Contents

1 Introduction **1**

1.1 Traditional Data Integration Techniques 2

1.2 An Alternative: Keyword Search-based Data Integration 3

1.3 Research Challenges 4

1.4 Contributions 5

1.5 Roadmap 7

2 Background of the Q System: Search-Driven Integration **8**

2.1 Search-Driven Integration: A Sketch 8

2.2 **Q** System Implementation 11

2.3 Data Model and Learning Model 14

2.4 Summary of Open Issues 19

3 Active Learning in the Q System **20**

3.1 Background 22

3.2 Finding Informative Queries 24

3.3 Ranking and Learning 29

3.4 Increasing Diversity in the Top-k 38

3.5 Incremental Update on Source Discovery 42

3.6 Experimental Analysis 43

3.7 Conclusion 57

4 Collaborative Learning in the Q System **58**

4.1 Combing Weights across Users 60

4.2 Constraint-based Collaborative Filtering 62
4.3 Collaborative Filtering-based Learning 65
4.4 Experimental Analysis and User Study 70
4.5 Conclusion 82

5 Interactive Query Debugging in the Q System 83
5.1 The Debugging Problem 87
5.2 Boolean Model 89
5.3 Generalized Weighted Model 92
5.4 Experimental Analysis 97
5.5 Conclusion 104

6 Related Work 105
6.1 Related Work in Data Management 105
6.2 Related Work in Machine Learning 109

7 Conclusion and Future Work 112
7.1 Summary 112
7.2 Directions for Future Work 113

Bibliography 116

List of Tables

1 Average number of edges receiving feedback per round, until full separation is achieved. . . . 76
2 Number of conflicting edges in datasets allowing multiple views. . . . 78
3 Learning rate: minimum number of feedback rounds to achieve separation of costs of gold edges and invalid edges. . . . 78
4 Minimum number of feedback rounds to achieve full precision & recall of answers to holdout queries. . . . 79
5 Excerpt from queries used in user study 81
6 User study results 82

7 Average number of labeling questions required to get a correct answer. . . . 101
8 Total number of true and false links confirmed, respectively, given the same amount of feedback on each dataset. . . . 102
9 Average number of labeling questions required to get a correct answer. . . . 103

List of Figures

1 Search graph with keyword search terms. In this example, the biologist wants to explore information related to the GO term “plasma membrane” as well as titles of publications containing this information. Each keyword may match a node with a *similarity score* and each pair of attributes may also match with a similarity score; this is captured by an edge *cost* c_{ci} that can further be broken into features with weights. Query results are comprised of *trees* whose leaves are the matching nodes; each result is given a cost equal to the sum of the edge costs. 9

2 Neuroscience search graph, for animal and human data uploaded from several sources. Metadata (tables, fields) nodes are white-on-dark, and data nodes are black-on-white. Keyword nodes (on bottom) are matched against both data and metadata, illustrated by dashed directed arrows. From these matched nodes, schema aligments (bidirectional dashed arrows) are inferred. Query answers are formed from *trees* linking keyword nodes. 10

3 **Q** System components: offline loading produces a search graph (with incomplete edges) and a keyword index of content. Given a keyword search, links are discovered as-needed via alignment algorithms. Top-scoring results are returned, according to feature weights that are learned based on collaborative filtering and user feedback. 12

4 Neuroscience data portal [55]: the user searches for “mayo seizure” and may give feedback on ranked results. Answers union and join across multiple sources (including the Mayo Clinic) and incorporate outputs of seizure detection algorithms. 12

5 Speed of learning to distinguish gold vs. invalid edges, without considering diversity 46

6 Speed of learning to distinguish gold vs. invalid edges: benefits of incorporating diversity 47

7 Separation for **Relevance** in the Bioinformatics dataset 47

8 Separation for **Mixed Ent** in the Bioinformatics dataset 48

9 Separation for **Clustering** in the Bioinformatics dataset 48

10 Separation for **Diversification** in the Bioinformatics dataset 49

11 Feedback rounds required to reach maximum recall, without considering diversity 51

12 Feedback on edges using **Relevance**: width is proportional to feedback steps; dotted lines indicate an invalid edge. 52

13 Feedback on edges using **EMC Ent**: width is proportional to feedback steps; dotted lines indicate an invalid edge. 52

14 Recall rate versus feedback rounds 52

15 Feedback rounds required to reach maximum recall: benefits of incorporating diversity 53

16 Quality of initial query answers 54

17 Quality of initial query answers 54

18 Diversification vs. clustering behavior with respect to feedback on Bioinformatics data ($k = 5$ for top-k, and lower value is better rank) 55

19 Diversification vs. clustering behavior with respect to feedback on IMDB data ($k = 5$ for top-k, and lower value is better rank) 56

20 Diversification vs. clustering behavior with respect to feedback on Mondial data ($k = 5$ for top-k, and lower value is better rank) 56

21 Graphical model for probabilistic matrix factorization (PMF) 69

22 Learning rate: feedback rounds to separate costs of gold vs. bad edges. 74

23 Costs of gold & bad edges after each feedback round, for **Co-MiRA** over **Bio**. Distribution of edge costs is indicated with error bars. 75

24 Costs of gold & bad edges after each feedback round, running **Co-MiRA** on **IMDB**. Distribution of edge costs is shown via error bars. 76

25 Average time spent on collaborative filtering versus overall query answering & learning, in each feedback round. 79

26 A worst case graph G_{worst} for the boolean model with n nodes. The left part has $n/2$ nodes and the right part has $n/2$ nodes. Either part is a clique. The left part and the right part are connected through a bridge. 90

Chapter 1

Introduction

Today innovation in data science provides the potential for increasing enterprise profitability and accelerating scientific discoveries. Due to the growing demands for “big data” exploration and analysis, data are now being collected in a broad range of application domains – thus are inherently heterogeneous in type and format. The huge promise of rapidly growing “big data” is driving greater needs for *data integration* techniques to link disparate sources and reveal valuable information. Data integration, therefore, is a generator of value in many aspects of modern society. For example, neuroscientists can build personalized medicine or diagnosis algorithms faster, if they could combine shared data from the scientific community. Search engines can better respond to question-answering style queries, if they could synthesize data from multiple sources on the Web. Policy makers in government organizations would have better resource for decision making, if they have access to more relevant data collected from different agencies.

Ideally, a desirable integration solution should support end users in the following aspects. First, the system should make it easy for end users to find and locate relevant data, i.e., data that meet users' *information need*. This also means a convenient interface for users to specify what information they are seeking and to refine retrieved results if necessary. Second, as data and meta-data evolve, such changes should be propagated to and reflected in integrated data as well, to keep the system up-to-date. Third, an integration solution should be scalable and easily maintainable over time, when new data sources are being constantly added. Finally, it should be able to serve a community of users, where user's interests may diverse in terms of information need and/or preference of data sources.

Example 1.0.1 *Consider the following scenario. Movie expert Alice would like to use the popular **IMDB** dataset, together with relevant entries with richer details in **DBpedia** (directors, actors, and places), as a whole. The underlying integration system maintains meta-data that specifies how to* map *elements from **IMDB** to **DBpedia**, and vice versa. She may be interested in finding movies that have certain actors, or more complex information such as how two actors are "connected" through common place of birth or common movie. Being a non-database expert, she would like to have an intuitive interface to specify her queries. These two datasets will change over time, as **IMDB** will contain new entries and **DBpedia** will extract frequently updated content from **Wikipedia**. In addition, another movie expert Bob may wish to use the same system. But he prefers to use entries in **YAGO** instead of those in **DBpedia**. The integrated solution should take this into account by promoting answers using entries in **YAGO**. In addition, in order to (semi)-automatically linking entities across different datasets, Alice or Bob may wish to specify how entities should be linked, e.g., due to similar names or same values for certain attributes.*

1.1 Traditional Data Integration Techniques

Great progress has been made on Enterprise Information Integration (EII [12]), where, due to central control and a well understood problem domain, standards can be defined and schema and data mappings can be developed. In general, the approach is to define one or more integrated or mediated schema capturing the data domain, use schema mappings [5] to map data sources into the mediated schema, and finally allow users to pose structured queries against mediated schemas.

A problem with this approach is that mappings are often created from combining outputs of schema matching algorithms [24, 32] and/or entity resolutions tools [28]. The resulting matchings are often ambiguous and can only be resolved with domain (commonly, human) expertise. As a result, many of today's tools aim for semiautomated matching, where the system makes predictions and relies on a human domain expert to correct any mistakes or resolve any uncertainties. Unfortunately, administrator vetting becomes a bottleneck to the system incorporating sources, so does the mediated schema itself as new sources may have concepts that do not yet exist at the global level. Moreover, EII techniques are not flexible in more open-ended, community-scale data integration settings for two reasons. First, it is generally difficult or impossible for the participants to agree on a single consensus data representation. Second, the needs of the community tend to evolve over time, making a single fixed schema obsolete very quickly.

In addition, there are also several shortcomings to having a database administrator inspect the output of a schema matching tool before adding the mapping to an existing system: (1) administrator vetting becomes a bottleneck to the system incorporating sources; (2) the metadata might not clearly describe the data that must be mapped[1]; (3) subtle variations in semantics may only show up in occasional, incorrect query results. Moreover, the mediated schema itself can be a bottleneck to adding new data, as new sources may have concepts that do not yet exist at the global level.

Finally, large-scale data sharing settings today pose problems beyond those commonly faced in the enterprise, where conventional Enterprise Information IntegrationII [12] techniques have been successful. Consider, e.g., the needs of the National Institutes of Health's "Big Data 2K" initiatives, which involve integrating data across scientific or medical fields, such as genomics or neuroscience, to build personalized medicine or diagnosis algorithms; or alternatively consider the Semantic Web problem of linking and querying Web tables, structured knowledge bases, and Linked Open Data sources on the Web. In these settings, open-ended data domains are constantly changing as new resources and concepts are created.

1.2 An Alternative: Keyword Search-based Data Integration

These observations have triggered a recent movement to tackle the data integration problem in a more *compositional* [38] and *iterative* fashion (sometimes termed "pay-as-you-go" data integration [22] or "dataspaces" [30]). Under this model, the objective is to immediately support queries over "partly integrated" data, and to enable the user community to drive

[1]Consider, e.g., the situation where users put data into comments fields because there was no appropriate column in the schema.

integration of the data that relate to their actual information needs. Over time, data will be gradually integrated.

While the pay-as-you-go vision has been well-articulated for some time, only recently have we begun to understand how it can be manifested into a system implementation. One branch of this effort has focused on enabling "schema independent" queries through *keyword search-based* (*search-driven*) data integration [77], where matches to individual keywords are assembled into query results by discovering "join trees" that link the matches. The model resembles keyword search over databases [88], but here the data is remote and links may be induced *on-demand* via schema matching and record linking algorithms. Under this model, the output of alignment algorithms is used directly to answer queries, with no administrator intervention: the system relies on the *end user* to have the domain expertise to vet the results, and to provide *feedback* [76, 77] on the system's ranking of (some) individual query results. Now instead of having a human administrator correct bad associations, the system must *learn the correct score* (possibly zero or infinite) of each individual association, given the user's feedback on query answers that are formed from multiple associations. Hence, the quality of search results can be improved iteratively.

Search-driven integration promises greater scale-up in the number of sources, since it leverages feedback on data quality from the *end-user community*, as opposed to a central administrator who may be a bottleneck. It has some overlap with recent work on crowd-sourcing for schema matching [89], entity resolution [82] and query answering [31] — which employ third-party workers (or Turkers) to link or fetch data. However, the search-driven integration model has some key distinctions: the users giving feedback are looking for answers to their query, thus are incentivized to help the system give good answers; the users are expected to have some expertise in the data domain, and thus can assess the plausibility of (many of) the answers.

1.3 Research Challenges

Surprisingly, there is little progress made on realizing this pay-as-you-go vision in *end-to-end* systems. A full implementation requires a set of methods to handle real world issues of uncertainty and diversity and poses several fundamental challenges.

(C_1) **Uncertainty of schema matchers:** User feedback and attention are a limited resource. By contrast, the number of potential answers can grow rapidly due to the number and complexity of sources, keyword matches and schema alignments, whereas the user only sees a screen of results. Given that we wish to disambiguate uncertain links from limited user feedback, how do we systematically determine what to present to the user and how to learn from limited feedback?

(C_2) **Diversity of user population:** In a community setting, users often have different information needs and goals as they pose queries, since they belong to different lower clusters: they may be seeking different kinds of data and may trust different data sources. As a result, they may give different feedback to the same query answer. How do we personalize query answers to each user in a community with different objectives?

(C_3) **Noise of automatically constructed datasets:** Due to noisy and incorrect alignments, keyword search algorithms may fail to give users relevant answers for a particular query, especially when amount of noise is overwhelming. In this case, how does the system interactively help users find relevant answers and remove bad links?

1.4 Contributions

This dissertation attempts to tackle the above challenges by developing a full keyword search-driven integration system that breaks large end-to-end integration efforts into user-driven, incremental sub-tasks. In particular, this dissertation aims to show that, combing expert guidance and machine learning can scale up many tasks in "pay-as-you-go" data integration. More precisely, this dissertation validates the hypothesis that search-driven integration can effectively

1. Help users identify relevant answers they have not previously seen and disambiguate irrelevant answers, even if they only see a screen of results (Chapter 3);

2. Combine and resolve different user feedback, even if user opinions diverse in a community (Chapter 4);

3. Assist users to debug incorrect answers by directing their attention to relevant data (Chapter 5).

Before diving into technical details, we briefly describe the key ideas we develop to validate the above thesis. First, for a user's keyword query, the screen of returned results is both the means for a user to get relevant answers and the place for integration system to obtain feedback. These are two separated and sometimes even competing goals. The key challenge is to select a set of results to present that balances user's information need and system's need of learning. To address, our approach is to estimate the "utility" of a given result to both the user and the system, base on which our system decides what results to show. In addition, serving different users' information needs in a community requires the system to *personalize* query results. To do this, the system *collaboratively* learns for each user what elements contribute to correct integration, combines and resolves such evidence, and discovers missing relevant elements for each user. Finally, during assisting user to interactively debug incorrect answers, the system attempts to repeatedly ask for and learn from informative feedback. The challenge is to generate on demand a minimum series of questions to ask, so that user can see correct results quickly. Our approach aims to quickly identify problematic links so as to reduce the amount of erroneous answers.

To summarize, this dissertation makes the following contributions.

1. We develop active learning techniques to repair schema alignments from small amount of user feedback, through uncertainty reduction and result diversification. The novelty here includes that of estimating amount of uncertainty associated with a query result and a scoring model for ranking and diversifying both useful and uncertain query answers.

2. We develop collaborative learning techniques to personalize structured query results from possibly conflicting user feedback in a community. The novelty here includes techniques for propagating feedback from one user to others and incorporation of collaborative filtering methods to generate user-specific answer-ranking strategies.

3. We develop interactive debugging techniques to identify where data experts could best improve integration. The novelty here includes methods for updating link clas-

sification under various models and for synthesizing informative examples, as well as prioritization strategies for selecting examples for user to inspect.

4. We develop a full implementation of the above techniques in the **Q** System prototype and conduct comprehensive experimental evaluation of the methods on datasets from different domains.

In most cases, we will leverage techniques from the machine learning community. However, directly applying them will be problematic, since the query trees we deal with have combinatorial structure. We will discuss this in more depth in corresponding chapters and in related work. Our application and adaptation of these techniques to address unique challenges in keyword search-based data integration is novel.

1.5 Roadmap

The thesis is organized as follows. In Chapter 2, we provide an overview of the keyword search-based integration model and review the **Q** System, a prototype implementation. We also define our data model and learning model in Chapter 2. We describe active learning techniques in Chapter 3 [86, 87], collaborative learning methods in in Chapter 4, and interactive debugging approaches in Chapter 5. We survey related work in Chapter 6. Finally, we conclude and point out directions for future work in Chapter 7.

Chapter 2

Background of the **Q** System: Search-Driven Integration

In this chapter, we review the background of model and setting for search-driven data integration, generalizing across several models [9, 11, 66]. This builds upon previous work [76, 77] but adds several modules that implement the proposed methods in this thesis. We begin with an example (Section 2.1), before presenting the prototype search-driven integration implementation in the **Q** System (Section 2.2). We then describe the data representation and learning components of our system model (Section 2.3). Finally, we summarize the open issues in the framework that this thesis addresses (Section 2.4).

2.1 Search-Driven Integration: A Sketch

We start with two examples. The first example consists of a search graph with keyword search terms in a bio-informatics application, illustrated in Figure 1. The second example is motivated by search-driven integration in neuroscience data sharing [55]. Figure 2 illustrates how data is extracted from user-uploaded files and added to a *content graph*.

Example 2.1.1 *User-contributed items are tracked in* ***Uploads****, and consist of a folder of files. One user uploads human data with PDF details (extracted into the* ***Patient*** *table) and a set of* ***EEG*** *traces in files. The EEG header gives unique IDs that include pathnames. A second user contributes data of animals with epilepsy, tracked in* ***Uploads*** *and the* ***Animals***

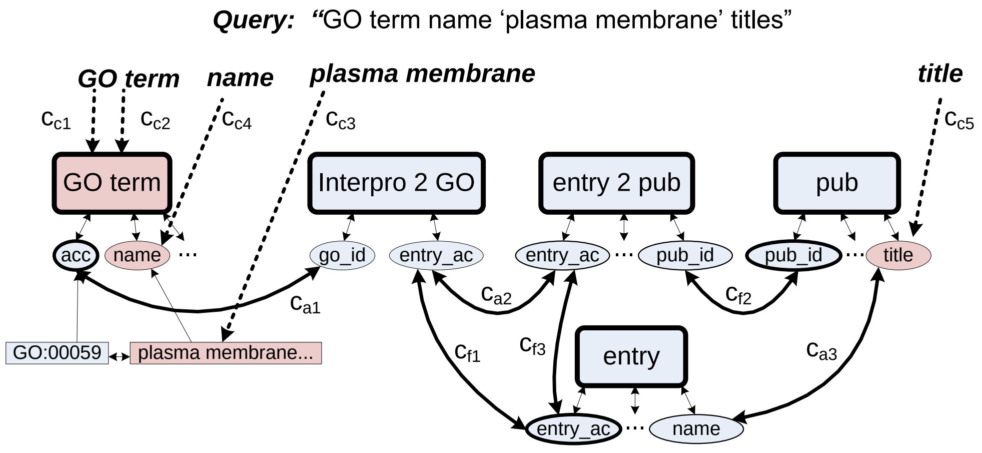


Figure 1: Search graph with keyword search terms. In this example, the biologist wants to explore information related to the GO term "plasma membrane" as well as titles of publications containing this information. Each keyword may match a node with a *similarity score* and each pair of attributes may also match with a similarity score; this is captured by an edge *cost* c_{ci} that can further be broken into features with weights. Query results are comprised of *trees* whose leaves are the matching nodes; each result is given a cost equal to the sum of the edge costs.

table, along with additional ***EEG*** *traces. A third user uploads a set of* ***Detected events*** *from externally running* **tool1** *over the EEG data. A fourth user uploads seizure predictions made by the* ***MHills*** *tool over EEG files.*

Nodes in the search graph represent relations, attributes, and values. Solid edges represent known links, including is-a *and* has-a *relationships, foreign keys, etc. Initially the dashed and dotted edges are not present as the* **Q** *System is not given knowledge of links among data and metadata items.*

Not shown in the figure, each node and edge is also annotated with *features* (sources of evidence for why we should believe the values obtained) that will be assigned weights, giving us a probability.

Suppose that a neuroscientist wants to find EEG data for patients with seizures detected in the *left temporal lobe*. To this point, we do not actually have links between all of the necessary data (no dashed lines), but the query itself can drive the system to do additional work.

Example 2.1.2 *The keyword query "left temporal seizure" will capture this. As illustrated*

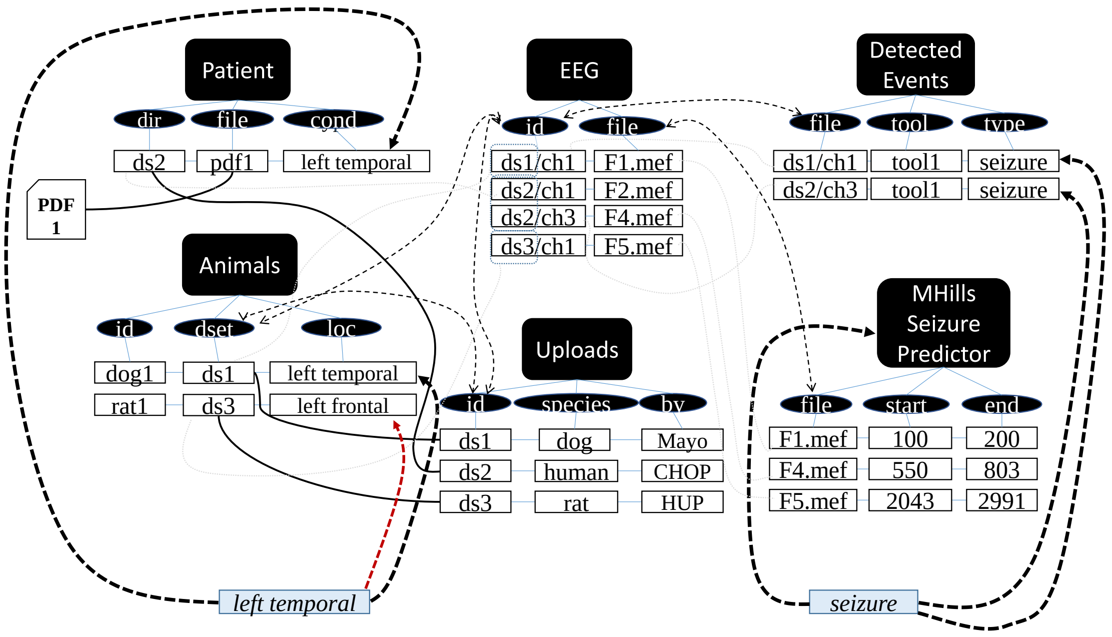


Figure 2: Neuroscience search graph, for animal and human data uploaded from several sources. Metadata (tables, fields) nodes are white-on-dark, and data nodes are black-on-white. Keyword nodes (on bottom) are matched against both data and metadata, illustrated by dashed directed arrows. From these matched nodes, schema aligments (bidirectional dashed arrows) are inferred. Query answers are formed from *trees* linking keyword nodes.

in Figure 2, we can encode search terms as nodes in the graph (at bottom), and match these (using string similarity metrics) against data and metadata nodes (directed dashed lines).

Now the system will seek to find trees *connecting the keyword terms, using a search process and schema/data alignment algorithms [24, 28, 32, 58] to find new links. These are illustrated by dashed bidirectional edges at the attribute (metadata) level (though value-level links are also possible). Each edge will be given a weighted* score, *and different candidate trees will be returned to the user in score-ranked order. Example trees include* ***Patient-Uploads-EEG-MHills*** *(selecting on* ***left temporal****),* ***Patient-Uploads-EEG-DetectedEvents*** *(selecting on* ***left temporal****),* ***Animals-Uploads-EEG-MHills*** *(selecting on* ***left temporal****), and* ***Animals-Uploads-EEG-MHills*** *(selecting on the substring* ***left****).*

The user provides feedback on the quality of the answers for these queries, from which the system learns to reweight evidence and recompute top-k answers. For instance, query answers including the ***Patient*** *table may be preferred to those including the* ***Animals*** *table;*

and the approximate-match to the substring "left" may in fact introduce incorrect data. The process iterates until the user is satisfied with all of his or her results scoring above some threshold or watermark.

Observe that search-driven data integration is distinguished from keyword search work by its ability to perform automatic linking on demand (using domain specific algorithms), and to incorporate the output of tools with uncertain output — followed by *learning to rank the quality of joined results* based on the user's feedback. Note that for users from different subfields, the notion of what constitutes a good or bad link, or a trustworthy or untrusted source, may vary. In the above example, our user preferred human data; but a second user in another lab may prefer animal data. It is not possible to mutually satisfy both users with the same scoring function.

Ultimately, for a given user, features on the bad edges and nodes in the search graph will receive poor scores (high *costs* that correspond to low probabilities), meaning that in effect the edges and noes will be removed from consideration in top-k query processing. Past work [76] has shown the model to be highly effective in distinguishing between bad and good edges, largely assuming feedback from *a single user with consistent feedback.* **This thesis goes beyond existing work by (1) incorporating *active learning* techniques to accelerate identifying good and bad edges, (2) considering *multiple users* who may not always provide feedback with the same goals in mind, and (3) developing *query debugging* mechanism to direct user to problematic graph regions.**

2.2 Q System Implementation

Now that we have seen the steps involved in search-driven integration, we discuss in more detail the system architecture required to support it. The core of this architecture, based on which this dissertation adapts, has been proposed and used in prior work on the **Q** System [48, 76, 77]. The overall architecture of the **Q** System, shown in Figure 3, can be divided into three stages, **where extensions to the second and third stages are main contributions of the thesis.**

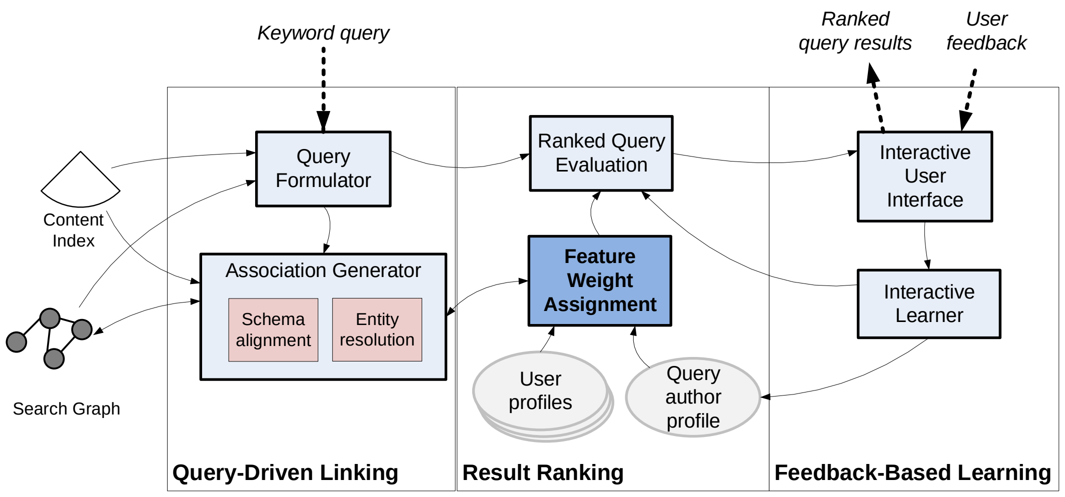
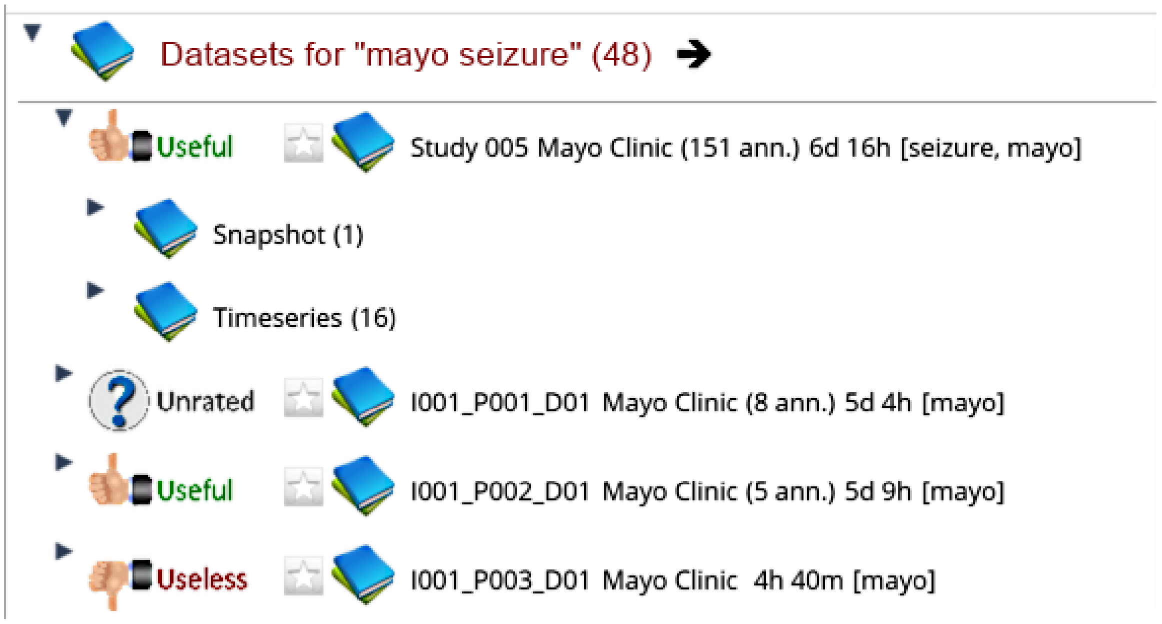


Figure 3: **Q** System components: offline loading produces a search graph (with incomplete edges) and a keyword index of content. Given a keyword search, links are discovered as-needed via alignment algorithms. Top-scoring results are returned, according to feature weights that are learned based on collaborative filtering and user feedback.



Datasets for "mayo seizure" (48)

Useful Study 005 Mayo Clinic (151 ann.) 6d 16h [seizure, mayo]

Snapshot (1)

Timeseries (16)

Unrated I001_P001_D01 Mayo Clinic (8 ann.) 5d 4h [mayo]

Useful I001_P002_D01 Mayo Clinic (5 ann.) 5d 9h [mayo]

Useless I001_P003_D01 Mayo Clinic 4h 40m [mayo]

Figure 4: Neuroscience data portal [55]: the user searches for “mayo seizure” and may give feedback on ranked results. Answers union and join across multiple sources (including the Mayo Clinic) and incorporate outputs of seizure detection algorithms.

1. Query-Driven Linking. The search graph starts off relatively sparse, in particular missing many candidate edges connecting data items obtained from different sources. Based on a combination of the past query workload and any current queries as obtained by the **Query Formulator**, the **Q** System's **Association Generator** runs a series of schema matching and record linking tools to discover new edges related to the concepts of interest. It does this in a way that *focuses* exploration of candidate edges to portions of the graph that are related to topics being queried [76].

2. Customized Result Ranking. Given a data graph and a set of edge and node weights, the **Q** System generates a ranked set of *Steiner trees* (minimal-cost trees connecting leaf nodes matching the keyword terms), whose results are merged to return top-k results. The **Ranked Query Evaluation** module uses an approximation algorithm for Steiner tree computation [77] to scale query answering to large graphs.

The actual assignment of weights for ranked query evaluation (the focus of Chapter 4), done periodically, is a contribution of this thesis: instead of simply using a set of weights per user (or a global set of weights), we develop alternative **Feature Weight Assignment** modules in subsequent sections. The task of this module is to take not only the user's past feedback into account, but also feedback from the rest of the user community, and to combine this to produce a set of weight assignments to be used in ranking answers for the current user.

3. Feedback-Based Learning. After it computes ranked answers, the **Q** System shows them in an **Interactive User Interface** (illustrated in Figure 4). The user is requested to mark answers he or she feels are *correct* and those that appear to be *incorrect.* The system uses these as training examples for an online **Feedback-based Learner** to adjust the scores on the edges (more specifically, weights for certain features that were used as evidence supporting the presence of the edge) in the particular results. The system will then refresh the results in the user interface, and the user may provide feedback for multiple rounds until the answer set looks good. **This thesis develops a set of active learning techniques [86, 87] (described in Chapter 3) that mixes high-quality answers with uncertain answers and a set of debugging methods(described in Chapter 5) that interacts with user to cleanse problematic edges and find satisfying answers**

when returned items do not meet user's information need.

2.3 Data Model and Learning Model

We now provide a more formal definition of the basic data model (Section 2.3.1), search procedure (Section 2.3.2), and learning model (Section 2.3.3) used in the **Q** System to produce a ranking of candidate query answers customized for each *specific* user. We start from models for **an individual user** then extend to multiple users. The models described in this section will lay the groundwork for how we incorporate active learning (Chapter 3) collaboratively learn from *all users* (Chapter 4), and interactively debug query results (Chapter 5).

2.3.1 Data Model: Graph, Features and Weights

The **Q** System's *search graph* encodes metadata and data items as nodes. Certain *association edges* are then added to capture relationships such as containment, subclassing, membership, and foreign key references. **Edge costs** depend on *features* and associated *weights*, upon which **cost of a query result** is developed. We give more precise definitions to these terms as follows.

Search graph: nodes and edges. Initialize the search graph to be $G = (V, E)$ where V is the set of nodes for each relation and attribute, and E is the set of association edges linking these nodes. The node set V will also dynamically add **keyword nodes** when user poses keyword queries. Each edge connects a pair of nodes and can be classified into the following four categories. (1) A *relation-attribute edge* connects a relation node and one of its attributes. These edges are derived from schema specification and will always be in the graph; (2) An *association edge*, or *alignment edge*, links two attribute nodes and represents a possible join. These edges have adjustable costs to measure the likelihood of a join, which will be learned over time (Section 2.3.3); (3) A *keyword edge* dynamically connects a keyword node to an attribute node, if an input keyword matches data in the attribute column. Such an edge also has associated *cost* to describe the quality of matching, described in Section 2.3.2; (4) A *node-node edge* represents (OO-style) subclassing or instantiation.

Features. The global **feature set** F contains several types of features. These features have weights, from which the *cost* of a query result (i.e. a Steiner Tree) is derived. Features include the following. (1) A *relation feature* identifies user's preference for (or bias against) a particular relation R, e.g., due to its authoritativeness or its relevance to his or her information need. This can also capture provenance of nodes, like the organization or author of a dataset. (2) An *alignment feature* describes evidence for attribute or data alignments. (3) A *keyword match feature* identifies keyword matching. We will describe in Section 2.3.2 how the **Q** System incorporates these features to perform search and ranking.

Feature vector. Given the features, every edge is mapped to a **feature vector**, which is a bit vector of binary feature values, where each bit specifies if that feature relates to the edge. This feature vector is associated with a **weight vector** we describe next. For a given edge, its participating features takes into account its type as well as the relations or keyword connected by the edge. For instance, given an alignment edge e connecting two relation attributes, its feature vector $\overrightarrow{\mathbf{F_e}}$ will set value 1 for its alignment feature and relation features and set value 0 for all the other features.

Feature weights and weight vector. Each feature in the global feature set has an associated **weight**, from which the **cost** of an edge is derived. The weight of a relation feature describes table authoritativeness; the weight of an edge alignment feature or a keyword match feature captures the quality of that matching. In our model, lower weights indicate better quality. Feature weights are *adjusted* through machine learning. We denote by $\overrightarrow{\mathbf{w}}$ the global weight vector where each element describes the weight value for a feature.

Edge cost and tree cost. Each feature f has a probability $0 \leq \Pr[f] \leq 1$ of being relevant or being a correct link (for alignment feature). This translates to the weight of each feature $w(f) = -\log \Pr[f]$, which is the *negative log likelihood* (and we will see why this is useful shortly). Each edge e has an associated **cost** $C(e)$, which is the *dot product* of its feature vector and the weight vector. This represents the negative log likelihood of the edge being relevant, since the dot product sums log likelihoods for features, assuming independence. This can be converted back to a probability by computing $2^{-C(e)}$. Like most keyword search-over-database systems [88], the **Q** System represents a query result for a given keyword query using a *Steiner tree* (explained in more detail in Section 2.3.2). For a given tree T, its feature vector consists of all features associated with tree edges and its **tree cost** $C(T)$ is the dot product of its **feature vector** and the **weight vector**. Again, the log likelihood interpretation is convenient, since it translates the product of probabilities in a tree to a summation of feature costs.

From metadata-level graph to data graph. To this basic metadata-level graph, we further add (on demand) nodes representing data values for individual attributes. These data nodes are linked to the attribute nodes in the search graph using *has-a* edges (from ellipses to rectangles in Figure 2, where "left temporal" is a data value for the **cond** attribute). Observe that we also link values from the same tuple.

Multiple users. We can naturally extend this data model to multi-user case by maintaining *personalized weight vector* $\overrightarrow{\mathbf{w_u}}$ for each user u. Search graph and features are shared by all users.

2.3.2 Query-Driven Linking, Search and Ranking

We now describe how the **Q** System ranks results for a keyword query by linking edges together, under this data model.

When the **Q** System is given a keyword query of the form $KQ = \{K_1, \ldots, K_m\}$, the *query formulator* first uses a keyword similarity metric[1] to match each keyword $K_i \in KQ$ against all search graph nodes (schema and data elements). It "overlays" onto the search graph a **keyword node** representing each K_i (see keyword nodes at bottom in Figure 2). It then adds a **keyword edge** from K_i to each graph node whose label matches the keyword with a sufficiently large similarity score. These edges are annotated with **keyword match features** and **weights** and have assigned costs. (Recall the dashed lines in Figure 2.)

The search procedure aims to *connect* keyword nodes together. It triggers an *exploration* of the search graph G, starting from graph nodes adjacent to each keyword node K_i. Explored nodes will be compared with other nodes in the overall search graph using *aligner* algorithms such as record linking or schema matching algorithms. If these predict a promising match, an **association edge** is permanently added between the nodes, annotated with **alignment features**. Feature weights, which predict alignment quality and will be adjusted through learning, are initially obtained from the outputs of existing matchers and alignment tools from the literature, such as [24, 28, 58]. The **Q** System currently uses

[1]By default tf-idf over the tuples in the data, although other metrics such as edit distance or n-grams could be used.

the primitive matchers provided by the COMA++ system [24], but could also directly use the final output from any matching tool.

In parallel with query-driven linking, the **Q** System explores the graph to return top-k query answers with minimum **tree costs** explained in Section 2.3.1. In the search graph, the **Q** System considers each tree with leaf nodes $K_1 \ldots K_m$ to represent a possible join query (each relation node in the tree, or connected to a node in the tree by a zero-cost edge, represents a query atom, and each non-zero-cost edge represents a join or selection condition). **Q** generates queries that may produce relevant answers by running an approximate top-k *Steiner tree algorithm* [77] to connect matching nodes in the search graph with the lowest-cost tree, and executes them and unions their results together in ranked order using a *top-k query processing algorithm* [46]. As discussed in Section 2.3.1, the cost of each query result tree is the dot product of its binary **feature vector** representing the set of participating alignments, and a **weight vector** representing the weight values for those features.

2.3.3 Learning from Consistent Feedback

In general the **Q** System's task is not finished once it has returned a set of query answers. Rather, the user may pose feedback over these results, by identifying good and bad results (we will formalize this shortly). We assume that the user looks over a portion of the k answers returned, and provides (1) feedback about which results are known to be incorrect, and (2) indirectly indicates a *watermark* separating the set of results verified from those that have not been inspected. In contrast to Web search, where the user generally only wants *one* valid answer and does not reuse query results, we expect here that the user wishes to keep the *set* of correct answers to a query, and that he or she may make the results *persistent* in the form of a view. Hence the user is incentivized to provide feedback.

The baseline model [77]. Assume that determining correct vs. incorrect results is context-insensitive (all users' feedback is consistent and can be combined). Then, the **Q** System can take a sequence of feedback, expressed as *linear constraints* on the (relative and absolute) costs of results, and learn to adjust the feature weights to satisfy those constraints.

Intuitively, each unit of feedback on a query answer tuple should give the system knowledge that a certain set of features have lower combined costs than some other set of features. Given that feedback is provided on *tuples*, yet some of our features are related to *queries*, we must be able to determine which query produced each tuple, and such that the feedback can be applied to the combination of query- and data-level features. The **Q** System tracks features associated to a tuple via *data provenance*: each query answer is obtained from running a union of conjunctive queries, which is in turn translated from a set of Steiner trees. Given two query results from one keyword query, one with positive feedback the other with negative, a "good" query result should have tree cost at least equal to the cost of a "bad" result, plus a minimum penalty (a "loss" explained below).

To incrementally adjust weights given user's feedback, our previous work [77] uses the MIRA learning algorithm [20], an online approximation to support vector machines. In a nutshell, MIRA attempts to find a new weight vector which is closest to the previous one and which satisfies constraints formalized from user's feedback. Since it will be adapted in this dissertation, we reproduce pseudocode for this as Algorithm 1. The algorithm takes input the search graph G along with the feature set F, stream of keyword queries $\{KQ^r\}$ for each point of time r, stream of user feedback $U = \{G_r, B_r\}$, where G_r is the set of good trees and B_r is the set of bad trees, given TS as the set of top-k Steiner trees for the current keyword query (k is a parameter). A key aspect of the MIRA algorithm is its reliance on a *loss function* that defines a penalty to assess to a tree that is "out of order" according to feedback (as described intuitively above). The default formulation is to use *symmetric loss* defined in terms of the number of non-overlapping edges between a given pair of query result trees:

$$L(T, T') = |E(T) \backslash E(T')| + |E(T') \backslash E(T)| \tag{2.1}$$

Observe from Line 7 of the algorithm that, given two query trees that are out of order, weights are adjusted to ensure that the difference between the cost of the trees satisfies the user constraint, with a cost differential of *at least* the loss function. The updated weight vector is closest to previous weights (Line 6) and ensures positive edge costs (Line 8). It can be iteratively found by a modification of the MIRA algorithm [77].

Algorithm 1 Online Learner.

Input: Search graph G, features F, stream of keyword queries KQ, stream of user feedback U, required number of query trees k

Output: Updated weights $\overrightarrow{\mathbf{w}}$

```
1: $\overrightarrow{\mathbf{w}}^0 = \overrightarrow{\mathbf{0}}$, $r = 0$
2: while $U$ is not exhausted do
3:     $r = r + 1$
4:     $TS = TopKSteinerTrees(G, \overrightarrow{\mathbf{w}}, k, KQ^r)$
5:     $(G_r, B_r) = U.Next()$
6:     $\overrightarrow{\mathbf{w}}^{(r)} = \arg\min_{\overrightarrow{\mathbf{w}}} ||\overrightarrow{\mathbf{w}} - \overrightarrow{\mathbf{w}}^{(r-1)}||$ s.t.
7:       $C(T_b, \overrightarrow{\mathbf{w}} - C(T_g, \overrightarrow{\mathbf{w}}) \geq L(T_b, T_g),\ \forall T_b \in B_r, T_g \in G_r$
8:       and $\overrightarrow{\mathbf{w}} \cdot \overrightarrow{\mathbf{f_e}} > 0\ \ \forall e \in E(G)$
9: end while
10: return $\overrightarrow{\mathbf{w}}^{r+1}$
```

2.4 Summary of Open Issues

To summarize, building upon architecture of the **Q** System proposed in the literature [77], this thesis targets the following open issues:

1. A more efficient feedback-based learner (in terms of labeling complexity that mixes high quality answers with uncertain answers to adjust feature weights with minimum amount of user feedback.

2. A feature weight assignment module that is scalable to a community of users to collaboratively learn from feedback provided by different users.

3. A more robust component of the interactive learner that guides user to debug incorrect results for a specific query.

Chapter 3

Active Learning in the Q System

The keyword-based data integration model enables the system and its users to focus their attention on those associations that relate to actual information needs. The associations relevant to frequently posed queries should be the ones that receive the most attention and refinement. In fact the pay-as-you-go approach can be used to complement and *inform* more traditional integration techniques: the keyword search log can help a human administrator determine which parts of the data to prioritize integrating, and provide clues for what mappings are most relevant.

However, to successfully learn to integrate data, the system must balance its need to acquire feedback useful for answering future queries, versus the requirement that each user immediately gets the information he or she needs. Today's keyword search systems have approached this problem by simply assuming the query scoring function is accurate: they return the top-k results according to the scoring function, which in turn bases its scores on the predicted (but possibly incorrect) output of matching tools. Under this model the user will attempt to remove false positives but has no way of seeing — and providing feedback on — false negatives.

Such a model works well when the system returns a good mix of correct and invalid results and the user can "separate" them. However, as the number and complexity of sources and their attributes increases, many potential queries are likely to have similar scores, due to inherent uncertainty in combining low-confidence results from various matching algorithms. The number of potential results can grow rapidly as the number of keyword matches increases, whereas the number of results seen by the user remains constrained by the dimensions of the screen and the limits of user attention. Thus, when a keyword-based data integration system selects queries to produce answers, it should not merely choose alignments based on the relative scores of associations — but also the *uncertainty* associated with a given query result, and the *informativeness* of feedback given on that particular result.

In this chapter, we use *active learning* to help the system determine which query results to present, given a combination of their predicted score, their inherent uncertainty, and the amount of information gained about other potential queries. Intuitively, the informativeness of feedback on a query result is related to how much uncertainty there is about the result's relevance to the query, and how many other *similar* share features with this result — meaning that feedback on the first result also reduces their uncertainty. We provide a more precise characterization of informativeness later in the chapter. Our work goes beyond previous attempts to use uncertainty-directed ranking in the pay-as-you-go-integration space, such as Jeffery et al. [49], which focused on individual mappings, by looking at the total uncertainty associated with *queries* and their results, and how this uncertainty should be combined with relevance ranking.

The key questions addressed in this chapter are how to estimate the utility of a given query to the system and to the user, and how to estimate the uncertainty of a query's score, in applying active learning to the problem of determining the relevance of associations to a query. Specifically, this chapter aims to make the following contributions:

- Techniques for estimating the uncertainty associated with a query, through the notions of entropy and variance, and by combining the probability distributions of the output for individual schema matching or record linking outputs.

- Pruning and active learning techniques that focus the user's attention on the query results most likely to either be relevant, or help the system produce better results.

- A scoring model using *expected model change* to relate the user's model of browsing data to how we should combine and rank both useful and uncertain query answers.

- A comparative study of two techniques to increase the diversity of results upon which feedback is provided: *clustering* similar join queries and choosing the most useful representative, versus *directly* selecting results for diversity.

- An experimental evaluation demonstrating and comparing the effectiveness of our approaches across several real data domains.

The chapter is organized as follows. Section 3.1 provides the context of our problem. Section 3.2 shows how we assess the *informativeness* of each query. Section 3.3 then describes how we combine informativeness and predicted score to return ranked query results, and to learn from feedback on them. Section 3.4 describes methods to increase diversity among top results (and hence improve the benefits of user feedback). We experimentally analyze our results in Section 3.6 and conclude this chapter in Section 3.7.

For simplicity, we present our work under the assumption that the data graph in the system is *static*, hence our goal is to return top-k answers from the graph as queries are posed and feedback is given. In reality, data in the **Q** system is dynamic (as new sources are discovered), and queries may be persisted in the form of dynamically updated top-k views. Hence we provide supplemental material in Section 3.5 explaining how this is achieved.

3.1 Background

The **Q** system encourages the user to "curate" the results of the query, distinguishing good answers from bad ones and establishing a preferred ranking order for the results. This leads to a tension between two desiderata: we must provide *some* relevant answers so the user is motivated to look through the results; but we want the system to continuously expand its ability to score new sources and new edges, i.e., increase its recall, meaning that we must also solicit feedback on results that include uncertain edges. These contrasting goals motivate the focus of this chapter: an *active learning* [68] approach incorporated into a component called the *suggester* module.

The **Q** system's suggester module ranks queries based on its uncertainty about their score, and how much feedback about their validity aids the system in predicting the score for other queries that have features in common with them. Its top results will typically be merged with the top-scoring query results, giving a mix of items for the user's inspection and feedback. Effective output from the suggester will help the system accelerate learning convergence while reducing the need for user intervention.

To achieve this, we incorporate the idea of active learning from the machine learning literature. Active learning improves the accuracy of learning while reducing the amount of training data: it relies on the ability of the learning algorithm to choose the data from which it learns. This is especially desirable for applications where labeled training data (in our setting, correct scores or costs for association edges, leading to correct query result rankings) is difficult to obtain. Typically, an active learning algorithm has access to an oracle and issues queries on unlabeled data. The oracle answers the learning algorithm by assigning a label associated with the query instance. In our setting, the user serves as the oracle.

The key question in applying active learning is how to select the next unlabeled instance for the oracle's annotation. A common approach is to adopt *uncertainty sampling*, a query strategy based on an uncertainty measure. This measure determines how uncertain the label given to the instance will be, and indicates how much extra information the underlying model may learn. In our setting, a computed query result is a sample, and its label indicates whether it satisfies the user's information need with the correct ranking. We develop a novel means of measuring a query result's uncertainty, given the uncertainty associated with its individual components like join associations.

We also explore another aspect: estimating how much feedback on a single tuple can help label a group of *similar* queries and their results. We develop a clustering strategy where queries sharing common edge and node structure are grouped together into one cluster, and a representative is presented to the user. Such clustering-ranking schemes have been previously used in other active learning applications, such as guided data repair [84] and record linking [4]. A major novelty of our work lies in determining and ranking the uncertainty of clusters of queries. We develop a measure based on the uncertainty score from the alignment(s).

3.2 Finding Informative Queries

In this section, we develop mechanisms for measuring the *uncertainty* of a query result, given knowledge of the uncertainty associated with the edge (schema match) and node (relation authoritativeness) components of the search graph. We then take into account the fact that queries may have overlapping edges or relations, meaning that feedback may benefit multiple queries. We seek to focus on returning "more informative" query results.

Our approach is to build over existing schema matchers and their underlying components. However, a challenge is that modern matchers [24, 25] combine the results of many *base matchers*, but return a single similarity score that does not reveal any information about how this was obtained. In order to determine the level of uncertainty associated with each potential alignment, we seek to estimate the *probability distribution* over the range of values, as suggested by [58]. To form this estimate, we compute a predicted distribution over all primitive matchers' scores for each association edge. We learn a weight for each of the base matchers, and use each weighted value as a point within a probability distribution for the possible values for the composite matcher. (The **Q** system learns how to best combine the weights from these base matchers, as described in Section 3.3.3.2.) Modeling feature weights as random variables enables us to estimate overall relevance of an edge, as well as the amount of uncertainty associated with it. This generalizes the model of our previous work [77], since the previous feature weight corresponds to a sure event with only one possible value and probability 1.

We consider in Section 3.2.1 how to compute weight distributions for the search graph using *features* of two types: (1) those suggesting attribute alignments (possible join *edges*) across relations on an attribute; (2) those representing authoritativeness or quality of relations or *nodes*. Section 3.2.2 then shows how such features' uncertainties can be *combined across multiple relations and joins* to get the uncertainty associated with a query (and its results).

3.2.1 Uncertainty in the Search Graph

We first explain how the estimated probability distributions for base matchers are incorporated into our search graph. We divide our discussion into the basic features associated with edges and nodes in the graph, and weights assigned to those features. These are built upon our simplified model in Section 2.3

3.2.1.1 Graph Components and Features

Extending our basic model described in Section 2.3, we treat the weights as random variables, to model the uncertainty of predictions of the schema and record alignment tools. For each alignment feature f_{AB}, we denote by $W_{f_{AB}}$ the random variable which maps a possible weight of the alignment feature to a probability value. Hence, $W_{f_{AB}} = w$ is the event that the feature weight takes value w. Similarly, there is a weight random variable $W_{f_{R_i}}$ for each relation feature $f_{R_i} \in F_r$. Thus, the cost of an edge $e = (A, B) \in E$, where A is an attribute of relation R_1 and B is an attribute of relation R_2, is given by the random variable

$$C(e) = \sum_i W(i) f_i^e = W_{f_{R_1}} + W_{f_{R_2}} + W_{f_{AB}}. \tag{3.1}$$

We now describe how features are computed for edges and nodes. (Our system also supports features whose values come from the data, e.g., score attributes within tuples [48, 76], and our model generalizes to this where each score attribute just becomes another weighted feature. For simplicity we focus on query rather than instance-based attributes.)

3.2.1.2 Edge (Schema Alignment) Features

The **Q** system encodes schema matches (attribute alignments) as *alignment features*, whose weights represent alignment qualities. For any two attributes A and B, the random variable $W_{f_{AB}}$ represents the score distribution we derive. We first focus on a single pair of attributes, where we treat the set of base matchers' outputs as an *ensemble* of classifiers, and let them *vote* on a prediction.

Schema Matcher Ensemble. The initialization procedure utilizes m matching primitives, i.e., base schema matchers. For any attribute pair (A, B), the i^{th} matching primitive algorithm produces a normalized discrete cost score $0 \leq s_i(A, B) \leq 1$. In our framework, as described before, a low score indicates high similarity.

Voting Heuristic. Since precision varies from different matching algorithms, we can assign each member a normalized preference $p(M_i)$, such that $\sum_i p(M_i) = 1$. This preference can be interpreted as the confidence level of a primitive. It also represents how heavily a primitive contributes to the final aggregated matching score. The values will be trained ahead of time, and will be learned (see Section 3.3.3.2). For any possible weight value $w \in [0, 1]$, we have the following estimation formula based on voting from the matching ensemble,

$$\Pr(W_{f_{AB}} = w) = \sum_{1 \le i \le m} \mathbf{1}(s_i(A, B) = w) p(M_i). \tag{3.2}$$

The above formula states that the probability of the alignment having a score w is the summation over all weights of matching primitives yielding the same score.

Relevance and Uncertainty. We can reason about both relevance and uncertainty of a particular alignment based on distributions over weight values for alignment features. For example, we can use the expectation $\mathbb{E}(W_{f_{AB}})$ to measure relevance, and then use the entropy of $W_{f_{AB}}$, $\mathbb{H}(W_{f_{AB}})$, or its variance, $\mathbb{V}(W_{f_{AB}})$, to measure uncertainty. We discuss this in more detail in later in this section.

3.2.1.3 Node (Relation Authoritativeness) Features

In some cases the user may have a certain preference for (or bias against) a particular relation R, e.g., due to its authoritativeness. We model this as a feature shared across all edges linking to the node R and its attribute nodes, and we initialize a uniform weight distribution for this feature.

3.2.2 Composing Uncertainty for Queries

In the **Q** system, keyword queries on the schema graph produce a set of structured queries, each generated from a Steiner tree T which is a subgraph of G. We combine uncertainty from each edge and define the uncertainty of a query by examining its corresponding Steiner tree as follows.

The cost of an edge $e \in E$ with features $f_1, f_2, \ldots$ and associated weights $W_1, W_2, \ldots$, where W_i is a random variable, can be calculated using Formula 3.1. Abusing the notation a little, we have the cost of a tree T derived from costs of all edges presented in T, as follows

$$C(T) = \sum_{e \in E(T)} C(e) = \sum_{i} \sum_{e \in E(T)} f_i^e W_{f_i}, \tag{3.3}$$

where $E(T)$ denotes the set of edges of tree T. We treat each W_i as being independent of the others. While in reality this may not be true, we will show experimentally that this heuristic is effective, and that it simplifies the learning procedure.

Consider a structured query plan modeled by a Steiner Tree T, we can infer relevance and uncertainty of the query from its cost expression. The **Q** system measures query relevance by the expectation $\mathbb{E}(C(T))$, and it captures query uncertainty using either entropy or variance. We describe each next.

Entropy. In information theory, *entropy* roughly represents the expected number of questions to be asked to decode a distribution. The entropy for a given random variable X is defined as:

$$\mathbb{H}(X) = -\sum_{x \in X} \Pr(X = x) \log \Pr(X = x).$$

However, since the distribution of $C(T)$ can be a set of possible values each with a uniform probability, we need more contextual information to derive more meaningful entropy values. Let $D = [s_{\min}, s_{\max}]$ be the domain of all scores, and $\{B_1, B_2, \cdots, B_b\}$ be the scoring "bins" which uniformly partition D into several ranges. Each B_i represents the range $[s_{\min} + (i-1)\frac{s_{\max}-s_{\min}}{b}, s_{\min} + i\frac{s_{\max}-s_{\min}}{b})$. Let G_j denote the event $C(T) \in B_j$. We have $\Pr(G_j) = \sum_c \mathbf{1}(c \in B_j) \Pr(C(T) = c)$. In this case, we consider the sample space to be all possible B_i. Hence, we can define the entropy value as follows

$$\mathbb{H}(C(T)) = -\sum_{1 \le j \le b} \Pr(G_j) \log \Pr(G_j). \tag{3.4}$$

In the **Q** system, given a tree T, we can compute its entropy by maintaining the distribution over total cost when traversing T. As Formula 3.3 suggests, when edge e is visited, we maintain the sum $SF(i) = \sum_{e \in E(T)} f_i^e$ for feature f_i. Finally, we can compute the cost distribution by independence assumptions on W_i, and therefore derive the entropy. This can be done by dynamic programming as shown in Algorithm 2.

Variance. Much like entropy, the variance value of a probabilistic distribution describes how diverse is the range of its possible outcomes. Using Formula 3.3 and independence assumption on different weights, we can compute the variance value as follows

$$\mathbb{V}(C(T)) = \sum_i SF^2(i) \mathbb{V}(W_i). \tag{3.5}$$

Algorithm 2 Computing entropy/variance for a query, or a tree

Input: A Steiner tree T

Output: Entropy and variance values for the total cost $C(T)$

```
1: for all feature $f_i$ do
2:    $SF(i) \leftarrow 0$
3: end for
4: for all edge $e \in E(T)$ do
5:    for all feature $f_i$ appears on $e$ do
6:       $SF(i) \leftarrow SF(i) + f_i^e$
7:    end for
8: end for
9: for all $d$ in the value domain $D$ do
10:    $P^0(d) \leftarrow 0$
11: end for
12: $P^0(0) \leftarrow 1.$
13: for $i \leftarrow 1$ to $|F|$ do
14:    for all $d \in D$ do
15:       $P^i(d) \leftarrow 0.$
16:    end for
17:    for all $d$ in domain $D$ s.t. $P^{i-1}(d) > 0$ do
18:       for all possible value $w$ which $W_i$ can take do
19:          $P^i(d+w) \leftarrow P^{i-1}(d)\Pr(W_i = w)$
20:       end for
21:    end for
22: end for
23: for all bin $B_i$ of total cost do
24:    $\Pr(B_i) = \sum_{d \in B_i} P^{|F|}(d)$
25: end for
26: return $H = -\sum_{B_i} \Pr(B_i)\log\Pr(B_i)$ and
    $V = \sum_i SF^2(i)V(W_i)$.
```

Given the various schemes for estimating queries' uncertainty, our next consideration is how to incorporate this uncertainty measure into a scoring function, such that we can rank the top-k answers to the user's query.

3.3 Ranking and Learning

The previous section showed how to estimate the relevance of a query (and its results). Once the **Q** system identifies the set of Steiner trees, it must rank them, such that the top-k results answer the user's initial query and maximize the utility of the potential user's feedback.

In this section, we consider two closely related issues. First, we need to *rank* the answers to a keyword query $Q = \{K_1, K_2, \ldots, K_m\}$, taking both relevance and uncertainty into account. We then consider how the system can learn from the user's feedback and update the scores and probability distributions associated with individual features.

3.3.1 Basic Ranking of Query Results

The cost of a query, i.e., a Steiner tree, derived in Formula 3.3, is a random variable from which our **Q** system determines its rank. A query's rank should depend on *relevance*, i.e., how likely is a query result to satisfy the user's information need. However, the query also has a certain amount of *uncertainty* in its score, which indicates how much extra information the **Q** system can learn from possible user feedback given on results derived from the query. A query's amount of uncertainty should also determine its rank since the user only sees a few top results and the system needs to maximize its learning gain. We have shown in the above section how to compute these two measurements for a given query. We now consider how to rank queries based on these values.

There is a tension between these two goals, rendering it semantically difficult to aggregate them for ranking and learning. We cannot directly use the decision-theoretical notion of stochastic dominance, nor skyline-based ranking, as these only produce *partial* orderings of results. Moreover, they fail to consider how the ordering affects the way a user provides feedback. Our trade-off between relevance and uncertainty in the long run is very similar to the problem of "exploration versus exploitation" in machine learning [68]. However, we must adapt existing techniques, because some of the key metrics are intractable to obtain in our setting.

We first consider two orthogonal notions of ranking, one based on the predicted relevance of results, and the other based purely on uncertainty, followed by a weighted combination of the two. Later in this section In Section 3.3.2 we will consider a more sophisticated means of combining the different facets.

Predicted-Relevance Ranking. A natural method of ranking is based on *predicted relevance*, i.e., the score obtained by combining the *expected values* of the features in the Steiner tree (query). Most keyword search-based systems, including prior versions of the **Q** system, adopt this ranking semantics. The final answer set in this model is a list of the k lowest-cost trees, in increasing order of expected cost. The top-k queries in this model can be computed by taking the graph, computing the expected cost for each edge and assigning it as the edge weight, then running a k-best Steiner tree approximation algorithm [77], which is tractable in practice.

Uncertainty Ranking. Conceivably, one could instead rank queries (and answers) according to the level of associated uncertainty. This is in some sense what systems supporting active learning typically do: focus the user's attention on the results that have the highest uncertainty, and thus the highest utility in learning how to rank.

The problem with this approach is that the entropy of a tree does not follow the principle of optimality with respect to the entropy of its subtrees. Due to this issue, we cannot directly apply previous methods [54], for enumerating and ranking top-k trees with respect to decomposable cost functions, as our uncertainty criteria are not decomposable. Although our algorithm for predicted-relevance ranking is similar to the Lawler-Murty procedure [54], there is no obvious way to determine the top-k trees with respect to entropy, without enumerating all trees. Moreover, since a tree's entropy is not directly related to its predicted relevance, the highest-entropy query answers may not be useful to the user. Similarly, query answers with high variance values may not help answer the user's information need. For these reasons we next consider a more feasible hybrid strategy we term *mixed ranking.*

Mixed Ranking. The notion behind mixed ranking is that top answers should include a predicted relevance component to try to satisfy the user's information need, while still including some uncertain answers that are useful from an active learning perspective. This can be achieved as follows. We first compute a large subset k' of the queries predicted to be relevant, e.g., the $2k$ most relevant query trees, and *then* choose from among these according to their uncertainty scores. Since we can tractably obtain approximate Steiner trees and compute entropy or variance for an individual tree, the overall mixed ranking is tractable in practice.

The scheme incentivizes the user to provide feedback: some of the query answers are likely to be of good quality, but they be mixed with bad answers. The user will be able to see that a small amount of feedback may result in an even more complete answer set.

Algorithm 3 Computing top ranked queries

```
Input: Schema graph $G$
Output: A ranked list of $k$ trees

1: Compute all top-$k'$ Steiner trees $\{T_i\}$
   w.r.t. minimum expected cost, where $k' \geq k$ is some set of candidate answers
2: (Optional; see Section 3.4.1) Cluster these trees into $k$ clusters,
   and choose for each cluster a representative $T_{r(c_i)}$
3: Select and rank the top $k$ results using one of the ranking methods, or a diversification
   scheme (see Section 3.4)
```

3.3.2 Ranking by Expected Model Change

The previous ranking semantics fail to consider *position bias*: the user typically examines results from top to bottom and stops at some point. This behavior suggests that top items are more likely to receive feedback. Even for the same set of results, different orderings may yield different amounts of feedback to the system. To address this problem, we present a novel ranking semantics using the notion of *expected model change* [69] and taking the user's browsing behavior into account. Briefly, the expected model change resulted from an unlabeled sample quantifies the estimated change to the current model *if we knew its label.* Typically, if a sample has a higher value of expected model change, the system is likely to learn more information if its label is revealed. Furthermore, our ranking by expected model change algorithm yields *a provable relevance guarantee*: we show a cost lower bound for the list of top-k results obtained from this ranking method. This is desirable because query answers need to satisfy the user's information need.

3.3.2.1 Browsing and Feedback Model

We adopt a user browsing model very similar to click models used in web search [37, 70]. The user starts by looking at the first result in the list and gives feedback on it. Then he or she continues to the second result with some probability p, or terminates his or browsing with probability $(1-p)$. If he does examine and give feedback on the second result, he will repeat the above behavior for the third result and so on. The user's browsing procedure terminates if he reaches the end of the list, or when he or she stops at a certain position. This stop position is what we refer to as the *watermark*, and we assume the user has vetted each result up to the watermark, and given negative feedback on any answers known to be incorrect.

Formally, if we let $r_1, r_2, \cdots, r_k$ be the top-k results where r_i is displayed at position i and let F_i denote the event that r_i is examined and feedback is given on it, we can model the user's browsing behavior as follows.

$$\begin{aligned}\Pr(F_1 = 1) &= \beta_1, \\ \Pr(F_i = 1 | F_{i-1} = 0) &= 0, \\ \Pr(F_i = 1 | F_{i-1} = 1) &= \beta_i,\end{aligned}$$

where

$$0 \leq \beta_j \leq 1, \forall j.$$

The second formula assumes that the user stops browsing if he does not examine the previous result in the list. The third formula quantifies the probability that the user continues to the result at position i if he has examined the result at position $i-1$. We generally assume that probabilities for continuing may vary for different positions. We also do not assume these conditional probabilities diminish as position moves from top to bottom.

3.3.2.2 Expected Model Change

Given the browsing model, we now formalize the definition for expected model change. This will allow us to quantify the aggregated amount of relevance and that amount of uncertainty associated with the top-k results. Consider the top results $\{r_1, \cdots, r_k\}$, where each r_i has a corresponding Steiner tree T_i, whose cost is a random variable. Informally, the expected model change for r_i is the expected amount of uncertainty reduction if r_i is given feedback to and no other trees in the top list receive feedback. The expected amount of uncertainty reduction can be computed by simulating our learning module, described in Section 3.3.3. The learning algorithm takes the list of top-k trees and feedbacks given to each tree (positive, negative or no feedback) as input, and changes probability distributions over feature weights in the schema graph accordingly. This results in change of cost variables for graph edges, and therefore the uncertainty measure of cost functions associated with the schema graph G. We denote by G^+ the new graph if r_i is given a positive feedback, and by G^- the new graph if r_i is given a negative feedback. We also denote by $\mathbb{U}(G)$ the total amount of uncertainty associated with schema graph G, obtained by $\mathbb{U}(G) = \sum_{e \in E(G)} \mathbb{U}(e)$, where $\mathbb{U}$ is a given uncertainty measure, for instance, variance or entropy. Similarly, $\mathbb{U}(G^+)$ and $\mathbb{U}(G^-)$ describe amount of uncertainty associated with G^+ and G^-, respectively. If we let ∇L_i be the amount of uncertainty reduction if r_i is given a positive feedback and $\nabla \overline{L_i}$ be the amount of uncertainty reduction if r_i is given a negative feedback, we can compute these values as follows.

$$\nabla L_i = \mathbb{U}(G) - \mathbb{U}(G^+), \qquad \nabla \overline{L_i} = \mathbb{U}(G) - \mathbb{U}(G^-).$$

According to the probabilistic interpretation of edge cost described in Section 2.3, we assume that $\alpha_i = 2^{-E[C(T_i)]}$ estimates the probability that r_i meets the user's information need, which in turn estimates the probability that r_i receives a positive feedback. Hence, the expected model change if r_i receives a feedback is given by

$$J_i = \alpha_i \nabla L_i + (1 - \alpha_i) \nabla \overline{L_i}.$$

Now that we have defined the expected model change for a given unlabeled query, we consider how to maximize the expected total amount of uncertainty reduction by incorporating our user browsing model. For a given query r_i, the user must first examine it in order to provide feedback. Hence, we can compute the **Q** system's expected utility from learning feedback given to r_i, denoted by $Y_s(i)$, as follows

$$Y_s(i) = \prod_{j=1}^{i} \beta_j J_i = \prod_{j=1}^{i} \beta_j (\alpha_i \nabla L_i + (1 - \alpha_i) \nabla \overline{L_i}).$$

Denote $B_i = \prod_{j=1}^{i} \beta_j$, which is the probability that the user has examined r_i. Similarly, we can obtain the relevance estimation, or the user's utility, for r_i, denoted by $Y_u(i)$ as

$$Y_u(i) = \prod_{j=1}^{i} \beta_j \alpha_i = B_i \alpha_i.$$

Combining utility scores for each individual result, we obtain the following two objective functions, one utility function for **Q** system's learning, and the other for the overall relevance.

$$Y_s = \sum_{i=1}^{k} Y_s(i), \qquad Y_u = \sum_{i=1}^{k} Y_u(i).$$

We apply an active learning strategy to maximize Y_s with respect to an ordering of a candidate result set, so that the **Q** system can learn as much information as possible. We will show that greedily ranking by J_i maximizes this objective function. Furthermore, we will show that Y_u has a lower bound if we rank results by J_i. This indicates that the total amount of relevance can be guaranteed.

Proposition 1 *Y_s is maximized if $\{r_i\}$ is ranked by descending J_i. Moreover, $Y_u \geq \frac{Y_s^*}{\sqrt{\sum_{i=1}^{k} J_i^2}}$, where Y_s^* is the optimal maximized value.*

Proof. We show that the above two statements hold for any given $k > 0$. The first part follows from the fact that $B_1 \geq B_2 \geq B_3 \geq \ldots$. To see this, assuming that there is an optimal solution Y_s' where $J_{i'} \leq J_{i'+1}$ for some i', we can obtain a new solution no worse than Y_s' by swapping $J_{i'}$ and $J_{i'+1}$. We can keep applying the exchange operation until $\{J_i\}$ is sorted. To prove the second part, we have

$$\begin{aligned} Y_s^* &= \sum_{i=1}^{k} Y_s(i) = \sum_{i=1}^{k} B_i J_i = \sum_{i=1}^{k} (B_i \alpha_i J_i / \alpha_i) = \sum_{i=1}^{k} Y_u(i) J_i / \alpha_i \\ &\leq \sum_{i=1}^{k} Y_u(i) J_i = \sqrt{(\sum_{i=1}^{k} Y_u(i) J_i)^2} \leq \sqrt{\sum_{i=1}^{k} Y_u^2(i)} \sqrt{\sum_{i=1}^{k} J_i^2} \\ &\leq (\sum_{i=1}^{k} Y_u(i)) \sqrt{\sum_{i=1}^{k} J_i^2} = Y_u \sqrt{\sum_{i=1}^{k} J_i^2}, \end{aligned}$$

which immediately yields the inequality.

The above proposition establishes a nice connection between exploration and exploitation: actively learning by the **Q** system still provides relevance lower bound. In practice, since computing the full ranking requires enumerating all Steiner trees, which takes exponential time, we instead compute a group of most relevant Steiner trees and rank them by their values of expected model change.

3.3.3 Learning from User Feedback

Recall that the keyword search-based data integration model identifies correct attribute alignments by learning from user feedback over query answers. In our **Q** system implementation, two modules — the interactive user interface and the feedback-based learner — enable this capability. Once a keyword query is issued, the **Q** system returns the results computed by the top-k Steiner trees[1], using one of the ranking algorithms. The system then converts these trees to conjunctive queries, executes these queries, and returns tuple answers [77]. The **Q** system displays resulting tuples to the user annotated with *provenance*, in the form of a tree describing the query or queries that produced the answer.

The user examines a portion of the result set of tuples and gives feedback, either positive (via explicit positive marking or by specifying the "watermark") or negative (via explicit negative marking). From the cumulative feedback, the **Q** system learns which features in the schema graph, i.e., attribute alignments and data source qualities, are most relevant to the user.

In reality, there are two sets of weight parameters that must be learned: those for correcting edge alignments, which take uncertainty and the weighted scores of base schema matchers into account (Section 3.3.3.1), and the weights that should be given by default to each of those individual base schema matchers, before further feedback is given (Section 3.3.3.2).

3.3.3.1 Learning Edge Costs

Our prior implementation for the **Q** system [77] incorporates the MIRA [20] online learning algorithm, an online approximation to support vector machines, to receive feedback from the

[1]For simplicity we describe the outcome as if each query produces one result, although the system actually iteratively enumerates top-scoring queries, even beyond k such queries, until it gets k answers.

user in a streaming fashion and to update weight values. In a nutshell, MIRA attempts to find a new weight vector which is closest to the previous one and which satisfies constraints formalized from the feedback.

The implementation in this chapter requires that feature weights be random variables instead of scalar values. The challenge, then, is how to adapt the online learning algorithm so that it can deal with probability distributions over feature weights. We adapt MIRA to our new setting as follows. The user indicates that a particular tree T^* should be the top ranked tree among the set of all top-k trees B. Let $\mathbf{w}$ be the vector where $\mathbf{w}_i = \mathbb{E}(W_i)$. We directly apply the MIRA algorithm, which takes the weight vector $\mathbf{w}$ as input and returns a new weight vector **w'** as output.

Finally, we compare $\mathbf{w}_i$ with $\mathbf{w'}_i$ for each i. If $w_i = w'_i$, then we keep the existing probability distribution for the weight on feature f_i, because f_i does not separate correct alignments and incorrect ones in the top-k trees and its weight does not need adjustment. On the other hand, if $w_i \neq w'_i$ for some i, the **Q** system will update this feature weight to a sure event: the weight random variable W_i will have value $\mathbf{w'}_i$ with probability 1. We show pseudocode for our learning algorithm in Algorithm 4, and the loop in Line 12 computes the new weight update. We use symmetric difference between two trees as the loss function, as in [77]:

$$L(T, T') = |E(T) \backslash E(T')| + |E(T') \backslash E(T)| \tag{3.6}$$

The online learning algorithm takes an *initial* score for an edge, based on a weighted combination of schema matcher outputs, and adjusts it. We next discuss how we set the initial score weights.

3.3.3.2 Learning Weights for Schema Matchers

The schema matching literature suggests that different matchers should have different weights in order to achieve a matching prediction with good precision and recall [24]. Such weight distributions may be different for different databases.

In the **Q** system, matcher weights are first applied to form probability distributions over alignment scores which compute query relevance and uncertainty. As the user poses more queries, the **Q** system learns the cost of individual attribute alignment. Under this model, it

Algorithm 4 Online learner

Input: Search graph G, user feedback stream U,
required number of query trees k
Output: Updated weights W

```
1: Initialize $W$
2: $r = 0$
3: while $U$ is not exhausted do
4:     $r = r + 1$
5:     $(S_r, T_r) = U.Next()$
6:     $w_i^{(r)} = \mathbb{E}(W_i)$
7:     $C_{r-1}(i,j) = \mathbf{w}^{(r-1)} \cdot \mathbf{f}_{ij} \;\; \forall (i,j) \in E(G)$
8:     $B = KBestSteiner(G, S_r, C_{r-1}, K)$
9:     $\mathbf{w}^{(r)} = \arg\min_{\mathbf{w}} ||\mathbf{w} - \mathbf{w}^{(r-1)}||$
10:    s.t. $C(T, \mathbf{w}) - C(T_r, \mathbf{w}) \geq L(T_r, T), \;\; \forall T \in B$
11:        $\mathbf{w} \cdot \mathbf{f}_{ij} > 0 \; \forall (i,j) \in E(G)$
12:    for all i do
13:        if $w_i^{(r-1)} \neq w_i^{(r)}$ then
14:            Update $W_i$ s.t. $\Pr(W_i = w_i^{(r)}) = 1$
15:        end if
16:    end for
17: end while
18: return $W$
```

may seem that matcher weights are no longer needed after initialization. However, consider the case where new data sources are added into the current system. In order to apply the same active learning module, we will still have to compute probability distributions over alignment features for new relations. The results of such estimations directly depend on matcher weights. Hence, we need to learn, and periodically re-learn, such weights in order to perform more accurate predictions for new data sources. Note that, however, the newly learned matcher weights should not be propagated to edges whose costs are already updated from user's feedback.

We periodically use a linear regression model to relearn matcher weights: we choose this learning method because the goal of learning the best parameter settings fits naturally into the regression setting. Alternatives like Naive Bayesian or SVM learning are more appropriate for classification tasks.

The aggregate matching score for an attribute alignment is a weighted sum of individual matching scores obtained from base matchers. Each alignment edge in the schema graph (after several rounds of learning edge costs) is a labeled sample, where the expected value of the alignment feature weight serves as the aggregated score. Formally, let $\{M_1, M_2, \cdots, M_m\}$ be the set of matches and $p(M_i)$ be the normalized matcher weight of M_i. For each attribute alignment edge $e = (A, B)$, the estimated aggregated matching, computed by $\mathbb{E}(W_{f_{AB}})$, is its label and we assume that the score correctly reflects alignment quality. Each matcher M_i produces score $s_i(A, B)$ on edge (A, B). The learning procedure aims to find $\{p(M_i)\}$ and to minimize the loss objective function

$$\min \sum_{e=(A,B)} ((\sum_i p(M_i)s_i(A,B)) - \mathbb{E}(W_{f_{AB}}))^2.$$

This is a classic linear regression problem, and we can invoke the learning procedure periodically.

3.4 Increasing Diversity in the Top-k

In active learning, the goal is to select query results such that the system maximizes its ability to learn (in this case, from user feedback). The uncertainty measure developed previously identifies the single sample (query result) with lowest confidence, in isolation. Similarly, the ranking schemes discussed effectively consider answers to be *independent* of one another. However, in the **Q** system, the user labels derived query results as positive or negative, and the feedback is converted into a modification of weights on individual features on edges or nodes. Some of these features may be **shared with other queries and their results**.

Clearly, this can lead to an issue: multiple closely related answers might have similar ranks by the metrics from the previous section. Yet feedback on any one of the answers might give most of the benefit of feedback on *all* of the answers.

Hence, in this section, we consider two approaches to increasing the *diversity* of the top-k answers, in order to get more beneficial feedback. We note that the diversity scheme should increase the number of different features present in the top-k answer set, such that the user has an opportunity to provide greater feedback. Yet (sometimes at odds with the previous goal) it should not suppress the best answers (particularly the most-likely-relevant answers) from appearing in the answer set. Finally, there should be some mechanism whereby the diversity scheme can be *dampened* as further feedback is given.

To address these goals, we consider two different schemes. In Section 3.4.1, we propose a clustering scheme, whereby we generate a large number of candidate results (in our experiments of Section 3.6, we generate $4k$), then we collect them into k clusters and choose a *representative* result from each cluster. In Section 3.4.2, we try an alternative approach, based on incorporating a diversity component into our ranking scheme, by extending one of the most effective techniques from the information retrieval literature.

3.4.1 Clustering Queries

Ideally, we can find a few "representative" query results to return in the top-k results, and learn about many other results' scores from these representatives.

Our clustering strategy targets this problem. It presents to the user the results of a query (Steiner tree) that shares some highly uncertain edges with other, also-highly-uncertain, queries — such that feedback given on results from the first query (tree) can also reduce the uncertainty of the other queries. To achieve this, we must estimate common uncertain information between two Steiner trees and how informative a given Steiner tree is with respect to a keyword query. We use these to cluster overlapping queries and choose a representative query per cluster.

Clustering Algorithm. Our clustering algorithm takes a set of query trees $\{T_1, T_2, \cdots\}$ as input and clusters them into k groups, one associated with each top-k answer, similar to the k-means algorithm. (Note that in our domain it is intractable to compute the entire set of query results and then perform hierarchical agglomerative clustering; instead we can only produce some partial set of results and return k of them. Hence k-clustering makes sense.)

We define the *center* tree T of a set of trees to be a tree where each edge e has an associated appearance frequency $r_T(e)$, which is the ratio of number of trees having e to the size of the set. For any general Steiner tree T, $r_T(e)$ is defined as $\mathbf{1}(e \in E(T))$. Now, we define the similarity between two trees as follows.

$$\hat{S}(T_1, T_2) = \sum_{e \in E(T_1) \cap E(T_2)} \mathbb{U}(e) \min(r_{T_1}(e), r_{T_2}(e)).$$

This similarity roughly estimates the amount of common uncertainty of the two trees by summing up uncertainty values on their common edges. It can estimate the similarity between two Steiner trees as well as similarity between a Steiner tree and a "center" tree of a set. We use standard k-means clustering over this similarity function to build clusters.

Choosing Cluster Representatives. Once we have a cluster of similar queries, the next key question is how to choose one from them for feedback, such that the **Q** system can learn as much information as possible. We determine such a representative based on a notion of *informativeness*. Intuitively, the informativeness of a query with respect to a cluster of queries measures how much uncertainty this particular query shares with other trees in the cluster. Formally, given a cluster of trees C and a tree $T \in C$, we define the informativeness as follows:

$$I_C(T) = \sum_{T' \in C, T' \neq T} \sum_{e \in E(T) \cap E(T')} \mathbb{U}(e) \tag{3.7}$$

where $\mathbb{U}(X)$ is any uncertainty measurement over a random variable X. Thus, we vote on a representative according to the above informativeness formalism, and choose the most informative tree to represent the cluster.

3.4.2 Directly Incorporating Diversity in the Score

The clustering approach of the previous section is well-aligned with our intuition that we would like to choose the "best" result from each group of related results. Yet there are some potential pitfalls to the approach. First, our choice of a representative result for each cluster may be poor: in this case, the user might see (and provide feedback on) a bad result, when other members of the cluster were good. Unfortunately the results of this feedback may down-rank all of the answers in the cluster. Second, it is not obvious how to gradually dampen the clustering scheme, such that initially we select with a heavy bias towards diversity, and after a few rounds of feedback we progressively de-emphasize the diversity component.

The information retrieval literature considers an alternative to clustering: there, the approach is to incorporate diversity *directly into the scoring function* [23, 27, 35]. Section 3.3 described how we can incorporate two different optimization criteria, namely predicted relevance and uncertainty, into the scoring function. Here we extend this even further, to consider diversity as a third optimization criterion.

A commonly used family of approaches in information retrieval comes from Gollapudi et al. [35], who proposed a series of functions to combine an existing score with a diversity measure. Of the proposed measures, we favor the one that chooses a set of results that maximizes the overall combination of relevance and diversification. Thus we adopt a scheme called *max-sum diversification*, which maximizes a linear combination of utilities of selected query answers and their pairwise distances. More specifically, we are given a universe U of objects to choose from (in our case, U is the set of all Steiner trees), the scoring function Y_{div} is defined over a subset S of k selected items from U as a weighted linear combination of any standard score metric from Section 3.3, plus a diversity component:

$$Y_{div}(S) = (k-1)\sum_{T \in S} Score(T) + 2\lambda \sum_{T_1, T_2 \in S} \hat{D}(T_1, T_2). \tag{3.8}$$

We can use the symmetric difference of the two trees as the distance function, which is defined as the number of edges in the union of the two trees but not shared by both of them.

Maximizing the above objective function is NP-hard [35]. Instead, consider the following equivalent reformulation [35]:

$$Y_{div}(S) \;=\; \sum_{T_1, T_2 \in S} D^{'}(T_1, T_2) \;=\; \sum_{T_1, T_2 \in S} Score(T_1) + Score(T_2) + 2\lambda \hat{D}(T_1, T_2)$$

This results in the Max-Sum Dispersion problem which has a 2-approximation [20]. We use this reformulation, but with the following adaptations in the **Q** System.

1. Unlike in many IR applications where the scoring function is mostly about relevance, we adopt our proposed ranking functions (for example, expected model change) so as to consider relevance, uncertainty and diversity together.

2. We also adaptively change λ. At the beginning, the system is actively exploring for more feedback. However, after the system has gathered certain amount of information, it will apply the exploitation strategy. To enable this, we can gradually decrease the value of λ to reduce the effect of the pairwise similarities (and hence that of diversification).

3.5 Incremental Update on Source Discovery

Algorithm 5 Incrementally add new data source

Input: Search graph G, subgraph representing new source $G^{'}$, keywords (K) associated with current view, cost function C, cost threshold τ.
Output: Augmented schema graph $G^{''}$,
with alignments between G and $G^{'}$.

```
1: $G^{''} \leftarrow G \cup G^{'}$
2: $S \leftarrow \emptyset$
3: for $k \in K$ do
4:     $S = S \cup GetCostNeighborhood(G, C, \tau, k)$
5: end for
6: for $v \in S$ do
7:     A = BaseMatcher $(G^{'}, v)$
8:     E$(G^{''}) \leftarrow$ E$(G^{''}) \cup A$
9: end for
10: Return $G^{''}$
```

In addition to finding informative query answers and learning from user's feedback, another challenge in keyword search-based data integration is to incrementally update the underlying model when we add new data sources [76]. This involves not only updating the base data in the form of the schema graph, but also updating any materialized views that were formulated through keyword search. We wish to automatically combine new data sources into the existing schema graph, and to predict edge costs in order to discover query trees to generate potentially useful new results for existing keyword search-based views.

In more detail, within the **Q** System, each keyword query can be saved as a view whose results can be revisited over time. For each view, we seek to only add new alignment edges that can potentially affect the results in the view, upon new data sources are connected. Formally, suppose we have $G = (V, E)$ as the existing schema graph and $G' = (V', E')$. We are also tied to a fixed view derived from a keyword search query $Q = \{K_i\}$. The goal of automatic incremental update is to derive a probability distribution over edge costs for each pair of attribute nodes (v, v') where $v \in V$ and $v' \in V'$. Notice that a naive way of performing such computation for all possible pairs requires examining $\Omega(|V||V'|)$ pairs, which is an undesirable quadratic term that does not scale well as the number of schema graph nodes becomes large. Ideally, we need a strategy to compute only that subset of possible joins that indeed produces results affecting the top-k answers of the existing view.

Our information need-based strategy adopts a pruning approach and limits our search space to only a subset of possible pairs. Let C_{max} be the maximum expected tree cost (relevance) among all top-k trees in a fixed view. We also set a threshold $\tau > C_{max}$ (but not too large). Building upon [76], we say that a new attribute node A is feasible if and only if there exists a keyword node $K_i \in Q$ such that the minimum expected cost from K_i to A is less than τ.

We formalize this in Algorithm 5, which identifies all feasible attribute nodes by using BFS. The algorithm starts with all the keyword nodes as seeds, and iteratively expands new regions. When a node is to be expanded, we check if its expected distance to the keywords is greater than τ. If this happens, the algorithm will stop further expansion from the node. After we obtain all feasible nodes, we will align each of them with every node in V to compute the costs.

3.6 Experimental Analysis

We now experimentally evaluate the different options for the suggester module, and their impact on learning, in the **Q** system. We seek to validate that active learning improves our ability to distinguish correct from incorrect schema mapping edges and return better query results, and to understand the differences among the ranking models, uncertainty metrics, and diversification schemes.

Datasets. A major challenge in conducting keyword search experiments across integrated data sources is that it is very difficult to identify the complete set of correct (“gold”) alignments, and even more difficult to identify the set of correct answers. To simulate this in a controlled yet realistic way, we focus on real data where the possible joins are known. We chose three well-curated datasets and removed information about foreign keys, meaning the **Q** system must use schema alignment tools to discover potential edges, and learning to improve its knowledge of the correct scores. The datasets were chosen to represent very different domains, and include the bioinformatics testbed used in [76], which combines the widely referenced Interpro and GeneOntology (GO) datasets; the popular Internet Movie Database; and the Mondial geographic encyclopedia. For each dataset, we choose a subset of the tables. Details about the datasets are shown below.

Name	Bioinf.	IMDB	Mondial
No. Tables	9	7	9
No. Attributes	48	21	36
No. True alignments	9	6	9
Size (in MB)	172	1082	7

Query Workload. For each dataset, we generated a workload comprising 7 (for Bioinformatics) or 10 (for IMDB and Mondial) keyword queries, whose results are revisited (and new feedback is given) three times. The queries were based on common-knowledge searches, and keywords with low selectivity were emphasized. Each of the visits (phases) is done in a randomly permuted order. Sample queries include “isomerase protein” for Bioinformatics and “Greece ‘health organization”’ for Mondial. Queries cover most possible join paths, including some with high and some with low uncertainty. For example, the path (roles.movie_id, movies.id) in IMDB has low uncertainty, while the path (interpro_interpro2go.go_id, GO_term.id) in Bioinformatics has high uncertainty.

Methodology. Experiments were conducted using our implementation of the **Q** system, which comprises approximately 55,000 lines of Java code. Evaluation was done using on an Intel Xeon CPU (2.83GHz, 2 processors) Windows Server 2008 (64-bit) machine with 8GB RAM, using JDK 1.60_11 (64-bit). For each dataset, the **Q** system first loads its schema (without knowing foreign keys) and constructs the schema graph. We run in parallel a set of schema matching primitives from the COMA++ schema matcher (Community Edition) to compute a weight distribution between $[0, 1]$ for every alignment feature. We prune potential edges for which all matching primitives give low similarity scores. We also assign one of 10 possible uniform weight distributions over $[0, 1]$ to each node (relation) feature.

Once the schema graph is constructed, we iteratively pose keyword queries. In each iteration, the **Q** system returns the top-k ranked Steiner trees for the keyword query, according to one of our ranking algorithms to be evaluated. Then we simulate the user’s feedback: a Steiner tree receives positive feedback if all of its edges are correct alignments (according to the actual schema information not provided to the system), and negative feedback otherwise. Note that our definition of a correct Steiner tree is very strict, as there might be additional useful alignments that are not specified in the schema. The **Q** system then uses the feedback to learn adjustments to the feature weights, and updates the costs of edges in the schema graph.

We initially consider two dimensions: the uncertainty metric, namely entropy and variance; and the means of scoring results from relevance and uncertainty, including using relevance only, mixed ranking, and expected model change. Combining these options, we look at predicted relevance (**Relevance**), mixed ranking using entropy (**Mixed Ent**), mixed ranking using variance (**Mixed Var**), expected model change using entropy (**EMC Ent**), and expected model change using variance (**EMC Var**). We then consider a third dimension, diversity: here we consider both **Clustering** and direct **Diversification** within the score function; we combine these approaches with the **EMC Var** ranking scheme, which proved to be the most effective.

Parameter Settings. We used $k = 5$ as the number of top queries to compute answers for each keyword search. To consider both relevance and uncertainty, we actually have the **Q** system fetch the top-$2k$ most relevant Steiner trees, and to choose from among these the top-k trees according to the combined ranking metric of study. When clustering, we use the top-$4k$ trees, which we combine into $2k$ clusters and then choose the top-scoring k results. We use the following COMA++ [24] schema matching primitives: data type similarity, string edit-distance, string q-gram distance, semantic similarity, and instance matchers. Initial matcher weights are trained offline before the experiment using a very small set of example attribute pairs. For diversification, we have used $\lambda = 0.9^{round-1}$ (where *round* is the answer-feedback step, starting at 1) for Bioinformatics and Mondial. For IMDB, we have used $\lambda = 0.7^{round-1}$.

We consider the following questions and present preliminary results

- Which active learning schemes most reduce the amount of training required to distinguish between gold and invalid schema alignments? Do clustering and diversification help? (Section 3.6.1)

- Which schemes require the least feedback to help the system discover correct alignments? (Section 3.6.2)

- Do the improvements in learning correct edges translate into an improvement in initial (before feedback) answer quality for keyword searches? (Section 3.6.3)

- For diversification, how does our clustering scheme compare with directly incorporating diversity into the ranking function? (Section 3.6.4)?

3.6.1 Speed of Learning

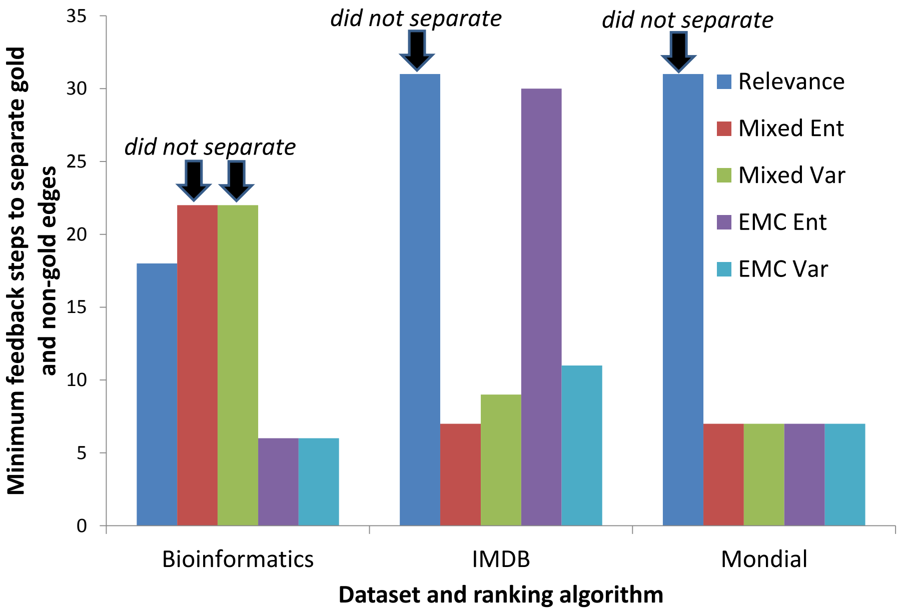


Figure 5: Speed of learning to distinguish gold vs. invalid edges, without considering diversity

The focus of our work on active learning is speeding up (in terms of feedback steps required) the **Q** system's ability to discriminate between valid (gold) and invalid edges. Hence in this first experiment we measure how many feedback steps are required to *separate* gold-standard and erroneous edges, where our test for separation is whether the intervals corresponding to the mean plus or minus one standard deviation, for gold-standard and erroneous edges, are non-overlapping and remain non-overlapping.

We performed a comprehensive set of experiments comparing the various algorithms across our three datasets. We summarize in Figure 5 how many feedback steps (rounds) were required to separate the intervals, for the various methods of combining uncertainty

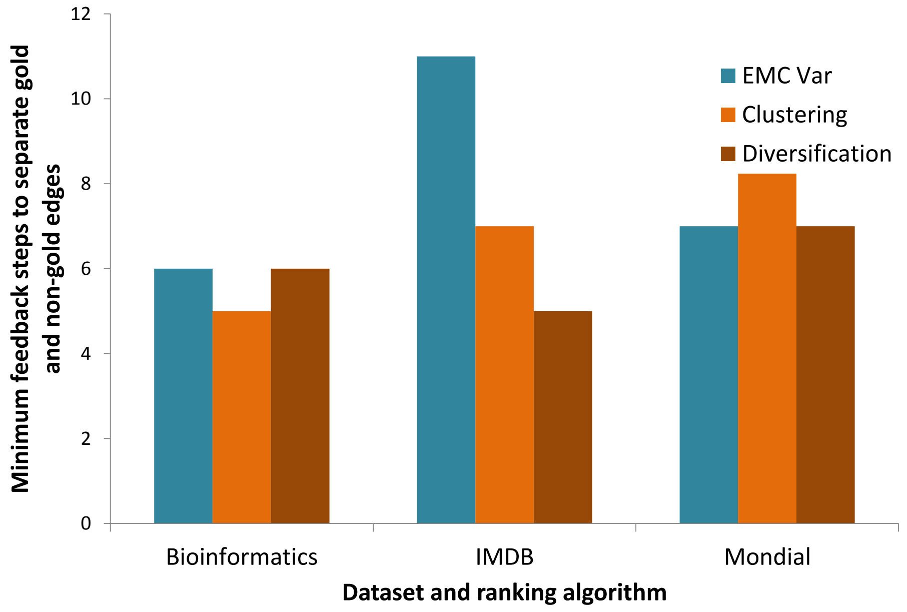


Figure 6: Speed of learning to distinguish gold vs. invalid edges: benefits of incorporating diversity

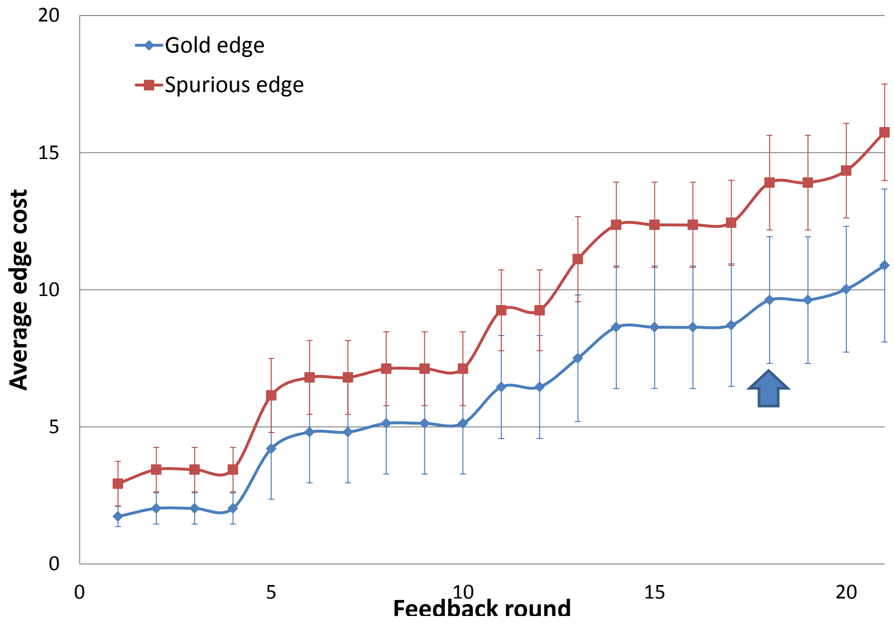


Figure 7: Separation for **Relevance** in the Bioinformatics dataset

and predicted relevance one answer at a time. Note that there are a maximum of 21 rounds for the Bioinformatics dataset and 30 for the other datasets. If separation was not achieved

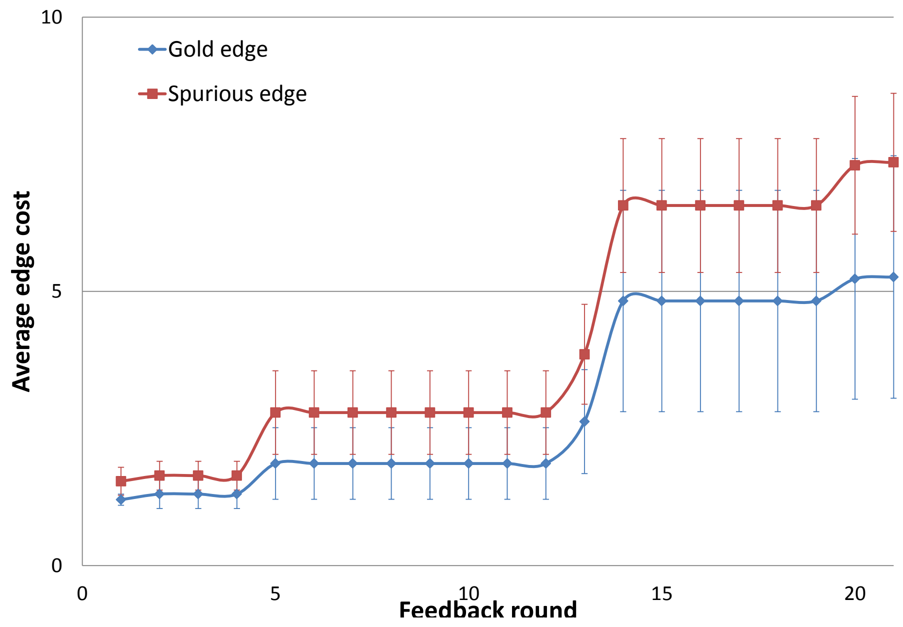


Figure 8: Separation for **Mixed Ent** in the Bioinformatics dataset

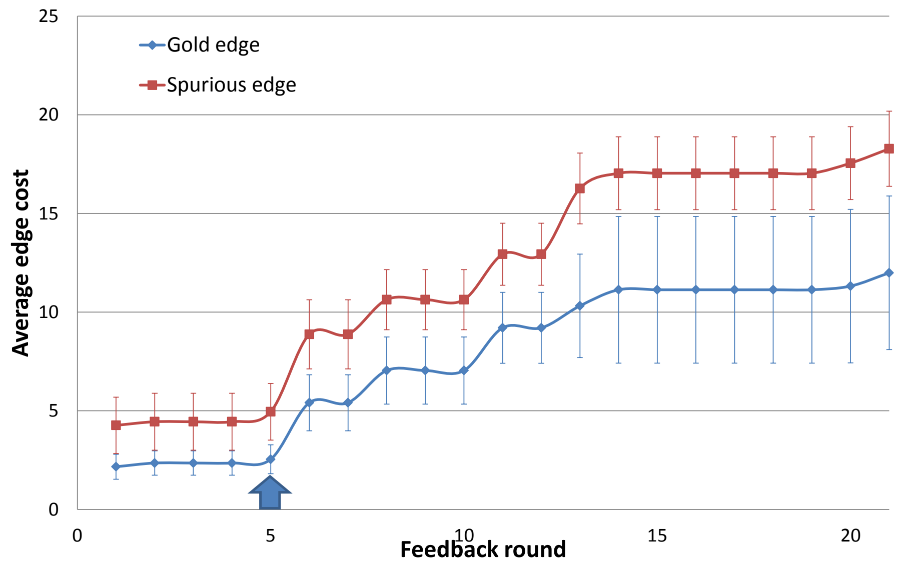


Figure 9: Separation for **Clustering** in the Bioinformatics dataset

within this limit, we mark this on the figure.

We see that the standard **Relevance** scheme achieves separation after around 18 steps, and in fact the mixed ranking schemes using entropy and variance, **Mixed Ent** and **Mixed**

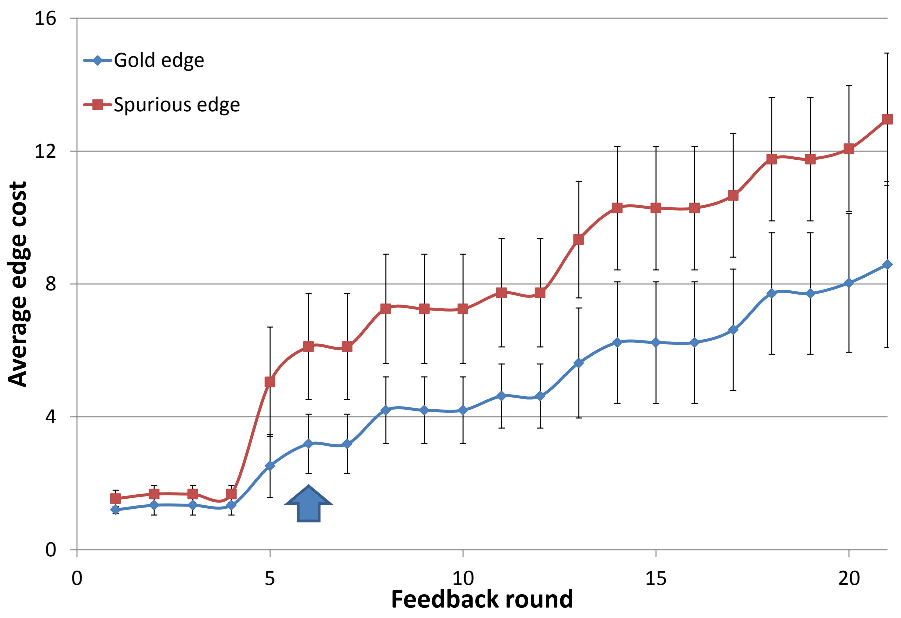


Figure 10: Separation for **Diversification** in the Bioinformatics dataset

Var, cause a drop in performance such that separation is not achieved within the maximum number of rounds. The expected model change schemes, **EMC Ent** and **EMC Var**, on the other hand, show dramatic improvement, achieving separation in 6 steps. For both IMDB and Mondial, the **Relevance** method does not achieve separation. With IMDB, **Mixed Ent** actually does best, with **Mixed Var** and its its expected model change variation **EMC Var**, also performing well. Surprisingly, expected model change plus entropy is ineffective here, given that entropy itself is useful. For Mondial, which has a relatively small set of edges, the various methods have equal performance.

Next, we consider how schemes for diversification change performance, in Figure 6. Here we use **EMC Var** as the baseline, since it was the most consistently effective method in the prior figure. We see that for Bioinformatics, the **Clustering** method speeds separation by a small amount (one step) versus the baseline and versus the **Diversification** scheme. For IMDB, the **Clustering** scheme provides significant benefit, but **Diversification** performs best by a notable margin. Finally, with Mondial, where there are relatively few edges, **Clustering** actually is less effective than the baseline algorithm. In contrast the **Diversification** scheme does not impede performance.

To give a greater sense of the differences in costs, we provide more detail for the Bioinformatics dataset, showing incremental change in each feedback round. This is representative of the other results, and focused on the target domain of the **Q** system. We see in Figures 7–9 the one-standard-deviation intervals around the mean score for gold-standard and invalid edges in the bioinformatics dataset, as each round of feedback is given to the system. (Recall that high scores mean high costs or dissimilarity.) In this dataset it takes 18 rounds to achieve separation of intervals for the **Relevance** method (Figure 7). The **Mixed Ent** method (Figure 8) never achieves separation within the 21 rounds of feedback (as its top-k answer set does not include results with several of the error-producing features).

The **Clustering** method (Figure 9) improves the situation, achieving separation after only 5 feedback steps. It does this in a relatively gradual way. The performance of the **Diversity** scheme is markedly different: for the first four rounds, the edge weights are relatively close in value (similar means, small intervals). In the 5th round, the values diverge more widely. There is a *tiny* overlap between the intervals around the gold and spurious edges (overlap of 0.05), and in the 6th round complete separation is achieved. Here we argue that the two diversification methods are essentially equivalent in performance, and as we saw previously, **Diversification** is strictly better (by at least 2 feedback steps) for the other datasets.

Although learning produces a large gap between the average costs of gold and non-gold edges, in a few cases, full separation is not achieved. Here the base schema matchers' scores for the incorrectly categorized edges have low variance and entropy — the matchers are "certain" about wrong alignments — so such edges do not show up in the top-k results and receive feedback.

Overall, expected model change with variance, with or without clustering or diversification, does best in learning to separate gold and invalid edges.

3.6.2 Recovering Gold Edges

We next study when active learning helps the **Q** system to find a more complete set of gold edges. The overall *edge recall* (ratio of gold edges to edges the system predicts) is determined by a combination of query load (which determines the set of nodes and to some extent trees) and ranking scheme (which determines feedback), and it may not always hit 1.0. For the diversity-agnostic methods, we show in Figure 11 how rapidly the system reaches its maximum recall value given our limited number of queries and feedback steps (21 for Bioinformatics and 30 for the others). We include in the captions, in parentheses, the maximum recall value achieved for each dataset.

On average, active learning (particularly the expected model change-based methods) significantly speeds up the rate at which the system achieves maximum recall. For Bioinformatics all of our new methods converge more quickly than **Relevance**, but the mixed ranking methods do not achieve separation and thus include a significant number of invalid edges. For IMDB and Mondial, the **Relevance** method is the one that does not achieve separation. We see little difference among the performance of the different active learning methods.

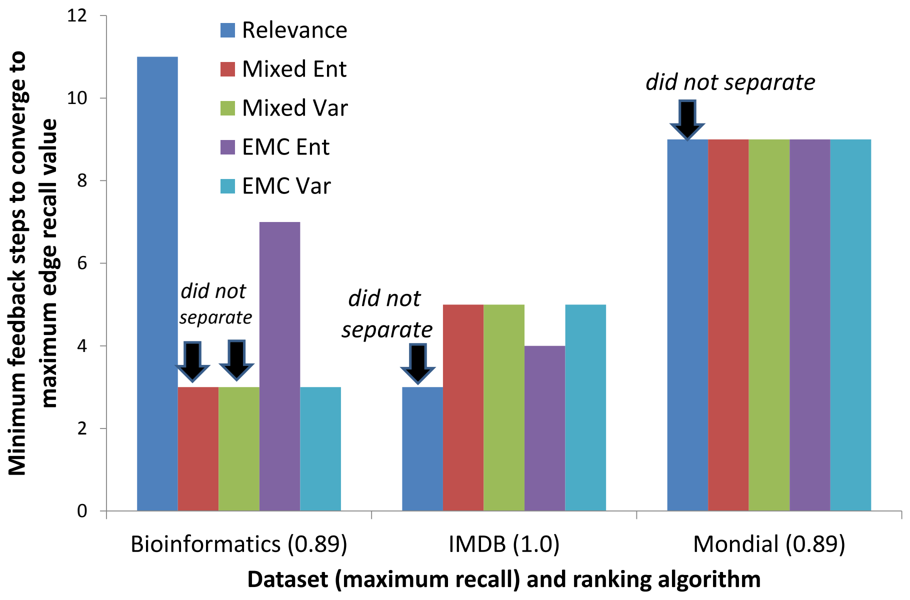


Figure 11: Feedback rounds required to reach maximum recall, without considering diversity

For more detail on the methods' behavior, we show in Figures 12 and 13 a visualization of a subset of the **Q** search graph, showing various relations and the edges among them. Dotted edges indicate invalid edges predicted by the schema matchers, and the width of the line indicates the amount of feedback given. We see that the **Relevance** method (Figure 12) provides less overall feedback on the edges, and explores fewer edges, than the **EMC Ent**

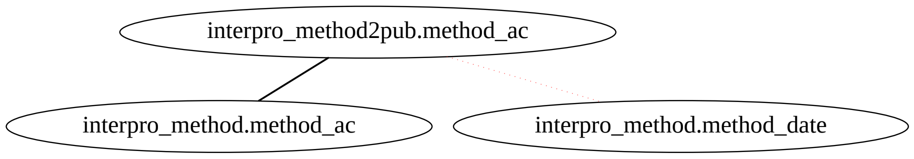

interpro_method2pub.method_ac
interpro_method.method_ac
interpro_method.method_date

Figure 12: Feedback on edges using **Relevance**: width is proportional to feedback steps; dotted lines indicate an invalid edge.

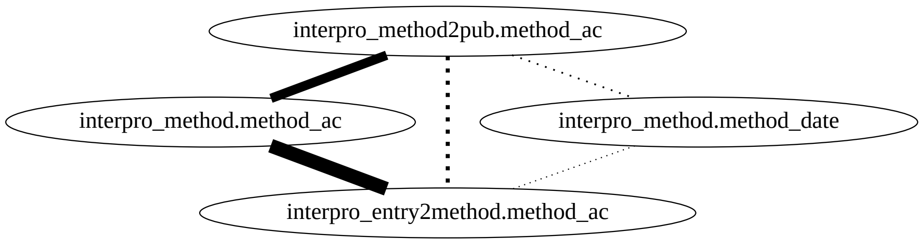

interpro_method2pub.method_ac
interpro_method.method_ac
interpro_method.method_date
interpro_entry2method.method_ac

Figure 13: Feedback on edges using **EMC Ent**: width is proportional to feedback steps; dotted lines indicate an invalid edge.

method of Figure 13. We conclude that expected model-change-based methods achieve the best combination of separation plus edge recall. To give an idea of how edge recall changes over time, we plot this for all three of our datasets versus feedback rounds in Figure 14. Observe that maximum recall is achieved within 5-9 rounds of feedback.

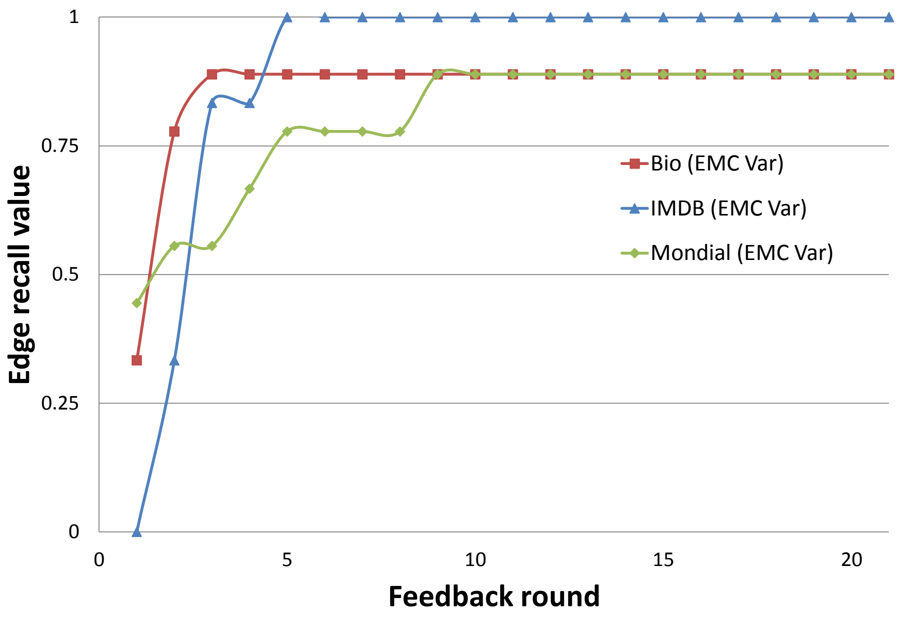


Figure 14: Recall rate versus feedback rounds

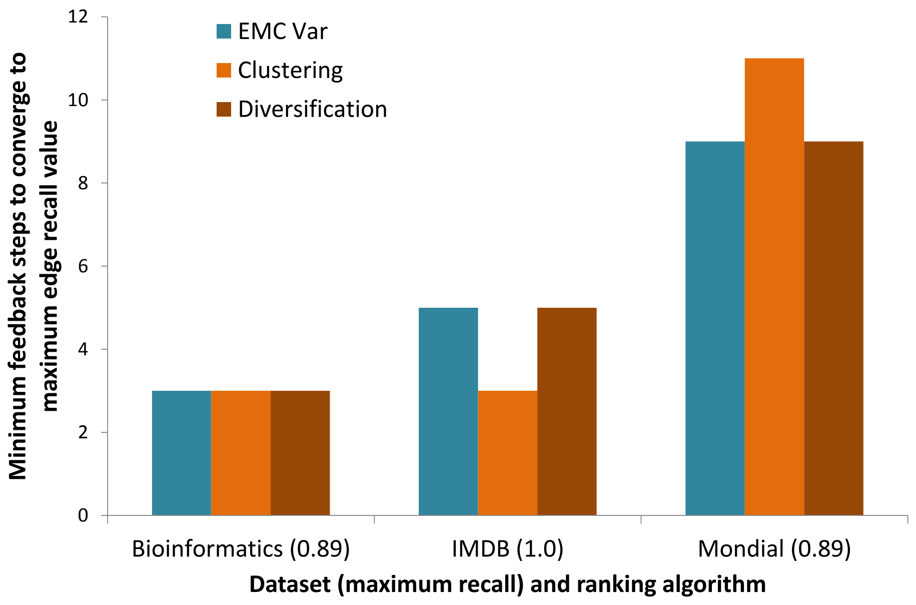


Figure 15: Feedback rounds required to reach maximum recall: benefits of incorporating diversity

Finally, Figure 15 shows the impact of incorporating diversity. The diversification schemes have no impact on the number of rounds required to get to maximum recall in the Bioinformatics dataset. For IMDB, the **Clustering** scheme actually achieves maximum recall in 3 steps versus 5 for the other schemes (yet it takes longer to achieve separation, see the previous experiment). For Mondial, the **Clustering** algorithm slightly impedes performance (as it did in the previous experiment).

3.6.3 Initial Query Answer Quality

While the system's goal is to learn correct rankings for the edges, the user's goal is to retrieve good query answers. Our final experiment measures how quickly the system — given feedback on the results from a set of *training* queries — can achieve 100% precision in the set of answers returned for a different but related *test* set of 5 queries. Figures 16 and 17 show the number of feedback rounds required over the training data; arrows over the bars indicate that full precision was not achieved even after the maximum number of steps. The figures shows that for the first two datasets, active learning makes a measurable difference, and that the **Clustering** and **Diversification** algorithms provide equivalent benefits over

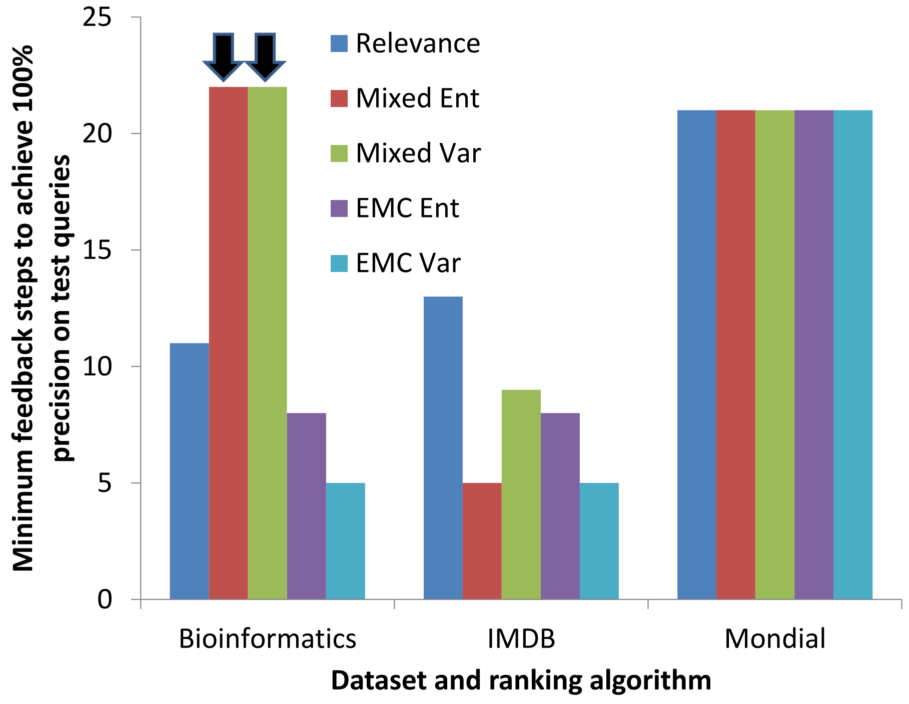


Figure 16: Quality of initial query answers

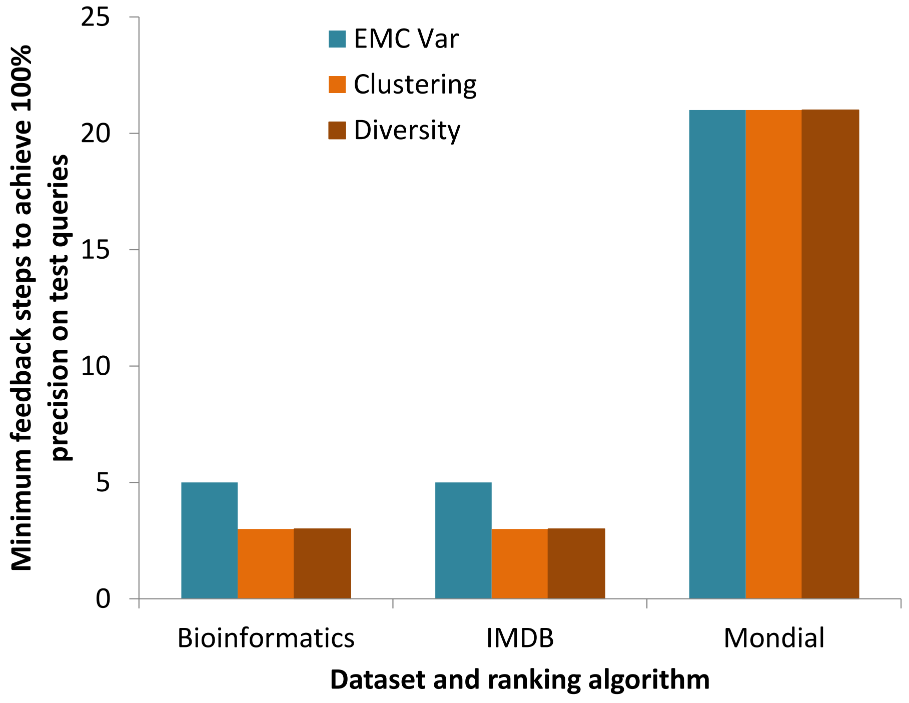


Figure 17: Quality of initial query answers

the expected model change methods. For Mondial the number of edges used in queries is small, so all methods receive adequate feedback to achieve perfect answers.

3.6.4 Clustering vs. Diversification

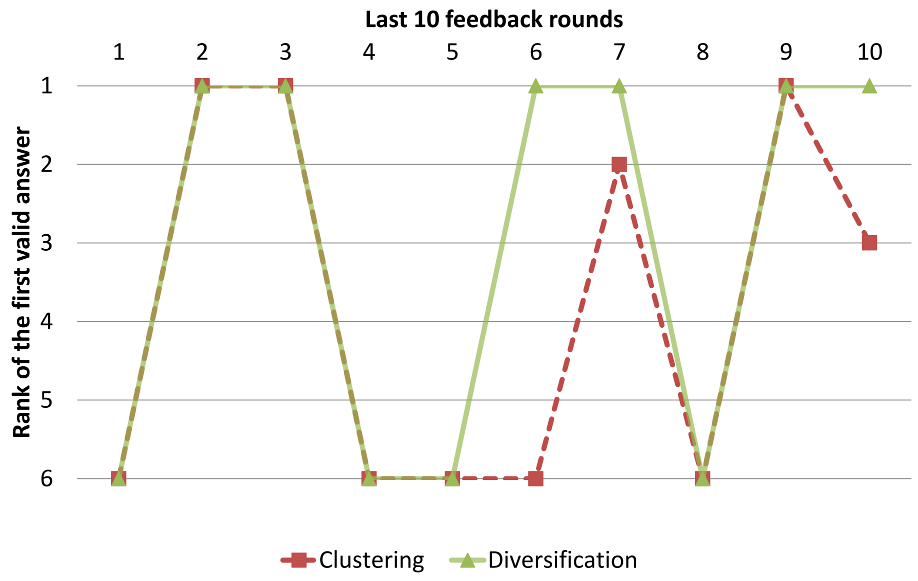


Figure 18: Diversification vs. clustering behavior with respect to feedback on Bioinformatics data ($k = 5$ for top-k, and lower value is better rank)

We have already seen from previous results that both **Diversification** and **Clustering** can help improve learning by a small but notable amount. Here, we further compare these approaches from the perspective of *stability*. Intuitively, we notice that the gradual change of λ in the diversification framework allows a "smooth" transition from heavily favoring diversity (aggressively gathering feedback from many different trees), towards a scheme with less emphasis on diversity (employing a more conservative strategy as more information is available to the system). With **Clustering**, however, since the system consistently explores a broader set of potential answers — even after several rounds, (1) the cluster representatives might not be correct and (2) correct answers may be affected by feedback on the (possibly incorrect) representatives.

We seek to experimentally validate this intuition by examining how stable the algorithms are at maintaining correct answers within the top-k answer set. In Figures 18, 19 and 20, we plot for each dataset, the rank of the **first valid answer** in the result set at each of the last

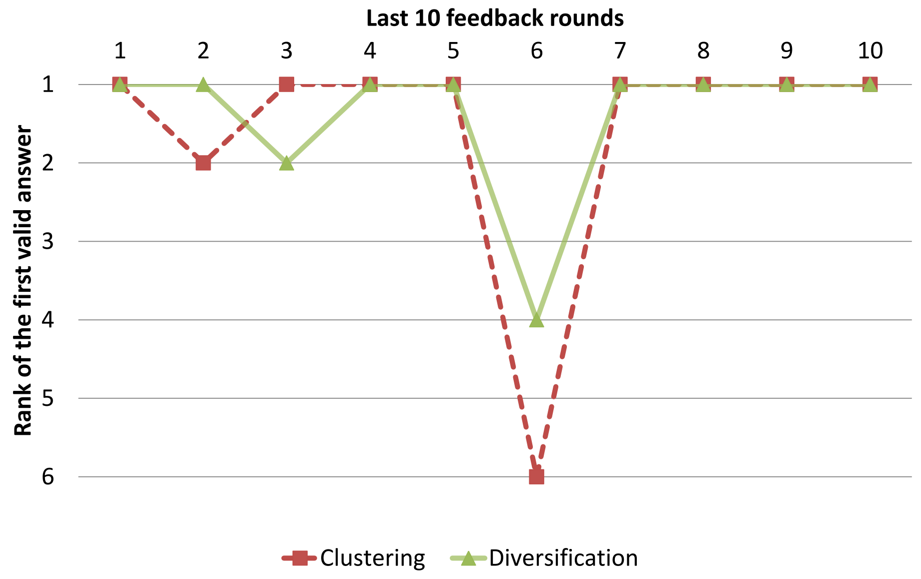


Figure 19: Diversification vs. clustering behavior with respect to feedback on IMDB data ($k = 5$ for top-k, and lower value is better rank)

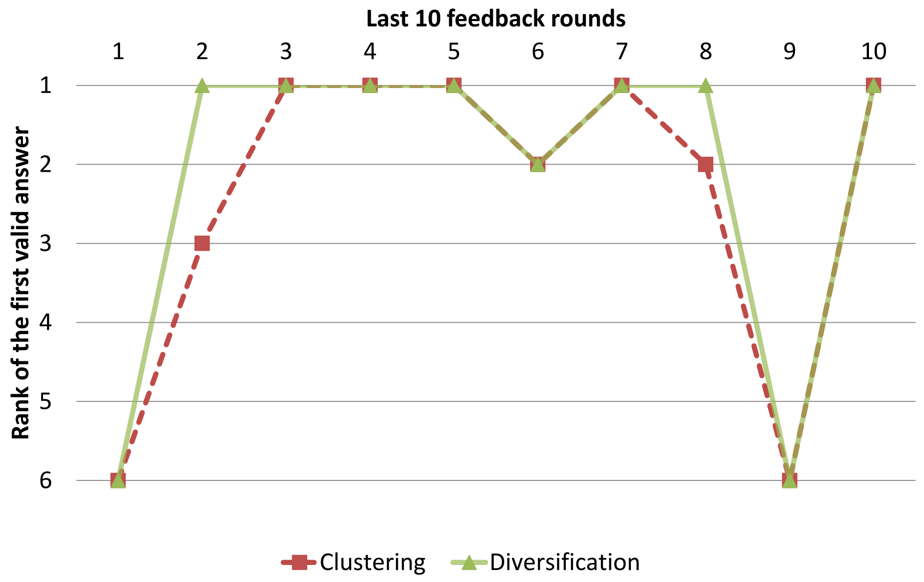


Figure 20: Diversification vs. clustering behavior with respect to feedback on Mondial data ($k = 5$ for top-k, and lower value is better rank)

10 rounds. Since we set $k = 5$ in our experiments, a rank of value 6 means that the top-k answers do not contain any correct tree. As we can see, for Bioinformatics and Mondial, **Diversification** is always more stable than **Clustering**: in any round, the rank of the first valid answer using **Diversification** is always higher than that using **Clustering**. In

particular, for the Bioinformatics data, at the sixth of the last ten rounds, **Clustering** fails to present any correct tree but **Diversification** does. This property also holds almost everywhere on the IMDB data, except for the second time point. Still, at the sixth round, **Clustering** drops off all valid answers but **Diversification** is able to find at least one.

3.6.5 Discussion

Overall, with respect to combining predicted relevance and uncertainty, we conclude that **EMC Var** is effective in virtually all scenarios, generally beating **EMC Ent**. Diversification generally does have the potential to speed up convergence by a small but measurable amount. Between the two diversification methods, we slightly prefer **Diversification** to **Clustering**, as it generally does not reduce performance versus the baseline, and it provides greater stability across feedback rounds.

While our active learning techniques show significant benefits, there remains room for improvement. We have assumed that the different base matchers' scores will be relatively uncorrelated, allowing us to detect uncertainty. Section 3.6.1 showed that sometimes, though, the base matchers do not provide enough score diversity to indicate potentially incorrect alignments. In the future we hope to study whether a greater diversity of base matchers would help.

3.7 Conclusion

In this chapter, we studied several techniques for incorporating active learning into keyword search-based data integration. The primary challenges were how to estimate the amount of uncertainty associated with a query built from edges induced during schema matching, and how to rank results in a way that maximizes utility to the user and the system, simultaneously. Our goal is a mix of relevance, uncertainty, and diversity. We showed experimentally that the notion of expected model change was highly effective.

Chapter 4

Collaborative Learning in the Q System

Search-driven integration leverages feedback on data quality from the *end-user community*, as opposed to a central data integrator or administrator. This makes it appropriate for community data-sharing efforts; we are seeking to validate our own efforts using real-world data and users from neuroscience [55]. Search-driven integration differs from crowdsourcing for schema matching [89], entity resolution [82] and query answering [31] in that the users giving feedback are those asking queries; and the users often have expertise in the data domain, and thus can assess the quality of (many) answers. Moreover, our users are *directly incentivized* to provide feedback to the system, as their goal is to assemble high-quality collections of (top-k) results to be analyzed.

Note that while we are performing learning over query results, our approach differs significantly from other work on learning conjunctive (or other) queries [14, 71]: we seek to learn from users how to rank query results produced through a Steiner tree matching algorithm, as opposed to learning the actual query expression in a particular language. Importantly, in scientific applications, **it is well known that the "ideal" ranking of results may vary substantially from user to user [15, 61]**, depending on his or her domain of interest, the task at hand, preferences for specific data sources, or confidence in particular alignment (e.g., record linking, gene sequence matching, coregistration) algorithms. This means that **feedback from different users may be inconsistent** — not because users are unreliable, but because they have different *information needs* and goals as they pose queries[61]. As we have applied search-driven data integration techniques to neuroscience and other applications with large user communities [55], we have found differences among users in:

- *Definitions of terms*: e.g., a "spike" in an electrical signal measured from the brain (e.g., EEG) often refers to a single "action potential" burst event to one subcommunity, but refers to a sustained event in another subcommunity.
- *Trust* in sources or tools: when different methodologies are used to record a value or classify a condition, different users may have different preferences.
- *Implicit goals*: one user may be looking for human data, and another for animal data; or one may be looking for EEG data and another may be looking for image data.

Therefore, a natural question is how to *combine* feedback from a community of users who may have different objectives, and also who may pose dissimilar queries and provide different amounts of feedback to the system. Due to the variations above, we require *personalization* of the query answers to each user, rather than *consensus* query answers [56, 82, 89, 91] or clustering of users based on overlapping queries [9].

Thus, this chapter investigates the problem of combining end-user feedback from heterogeneous users of search driven integration systems. More specifically, the feedback is over results generated from keyword searches converted into Select-Project-Join-Union queries over uncertain edges and nodes. An example application is the neuroscience data sharing and search portal being developed by our group (see Figure 4 for an example of a set of query results, using data from the world-renowned Mayo Clinic).

The task of combining feedback on a set of items from different users, to predict *missing* feedback for any item-user pair, has been extensively studied in recommender systems [51] and in the area of collaborative filtering (CF) [73]. Yet our problem setting is different and requires novel solutions. First, we are dealing with the *combinatorial* nature of query answers on which users give feedback: answers may consist of multiple data items and joins among them. We must learn the quality of each *individual* item and each possible alignment (as judged by a particular user). Second, we seek to *combine* online learning and collaborative filtering aspects into a unified framework that produces effective rankings for users with different preferences. We make the following contributions:

1. Formal semantics for combining and resolving conflicting feedback on structured query answers from end-users, to learn to better integrate data.
2. Techniques to combine *training examples from feedback* from multiple users, to learn a custom answer-ranking strategy per user.
3. Alternate techniques that take *learned weights* from each user and combines them using collaborative filtering techniques, to generate user-specific answer-ranking strategies.
4. Extensive experimental validation, over **real data** in several domains, to compare our strategies and demonstrate that they effectively combine and resolve feedback from multiple users.
5. We confirm our experimental conclusions through a **real user study with neuroscience data**, in which users have different preferences in the kinds of data they seek.

The chapter is organized as follows. Section 4.1 outlines the collaborative learning functionality that is the focus of the chapter. We describe in detail our two families of collaborative filtering techniques in Sections 4.2 and 4.3. We report experimental results and user study in Section 4.4. We conclude this chapter in Section 4.5.

4.1 Combing Weights across Users

In the **Q** System, each user is motivated to provide feedback, such that he or she obtains a set of top-scoring answers to match his or her preferences. A key issue is how to "bootstrap" a user's preferences from limited (even zero) feedback, by taking into account the feedback of other users. At first glance, it seems intuitive to combine feedback by using majority voting methods (as is commonly done in crowdsourcing to reduce erroneous inputs [31, 82]) to get consensus weights. For example, given a feature, one can aggregate all weights learned from each individual user through averaging. However, consensus weight values may not be ideal for helping find the "right" search results for a particular user. Alternatively, a clustering method was recently proposed [9], but it relies on users giving significant amounts of feedback over common data, which has a "cold start" problem. Instead, we seek to combine feedback across a community of users with different data expertise, varying amounts of feedback, and conflicting data interpretations.

Problem definition. This motivates the focus of this chapter: *collaborative learning* approaches incorporated into the **Feature Weight Assignment** module (Section 2.2). The task is to produce for each user u a *personalized weight vector* $\overrightarrow{\mathbf{w_u}}$ (Section 2.3.1) that captures custom cost values, as well as any "unknown" costs. Here we assume that each user focuses on a region in the search graph as her "area of expertise", where the user defines her own preferences over data sources and links. At any point of time, each user can pose keyword queries and can give feedback onto query results, based on her preferences. This gives us a stream of queries and feedback from the user community $\{u_i\}$. The goal of collaborative learning, given the above stream, is to incrementally find a personalized weight vector $\overrightarrow{\mathbf{w_{u_i}}}$ for each u_i, such that the following criteria are satisfied in the long run:

1. Within a user's area of expertise, personalized feature weights should be favored for her preferred features versus other features, even if other users assign differing scores.
2. Outside a user's area of expertise and feedback, personalized feature weights should reflect others' feedback.

As mentioned previously, the challenges lie in carefully selecting the approach and algorithms with an eye towards both running time (fast response time at scale) and effectiveness — since the collaborative learning algorithm must be periodically invoked as part of an online learning process. In this chapter we investigate two strategies for customizing the weights for the personalized search graph.

Strategy 1: *Reformulate* the online learning problem. As discussed previously, the **Q** System translates each item of user feedback to a linear constraint on the scores of query results. Naturally, we can look at the union of all such constraints expressed by a group of users. The issue with this approach is, with conflicting feedback among the users, the system will be *over-constrained.* However, if we have a *measure of similarity* among the users, in the online setting we can update the set of weights for user u by merging other users' feedback with U's feedback via scaled *soft constraints* [64], whose violation should be approximately minimized. We can associate each soft constraint with a *penalty* derived from *user similarity* between u and the other user. Such an approach allows us to incorporate the notion of user similarity and fits very nicely into the underlying optimization problem at the heart of the MIRA algorithm. We consider this approach in Section 4.2.

Strategy 2: Combine the *learned* weights. The above approach alters the internal representation of feedback as provided to the online learning algorithm. Alternatively, we can adapt techniques from the literature on collaborative filtering [73], commonly used to make personalized recommendations on the Web, to instead formulate a *missed value prediction* problem. Here we decouple the filtering problem from the particular online learning algorithm for adjusting alignment cost values. In this approach, our collaborative filtering algorithm periodically takes the user u's learned weights, and "overlays" weights for alignments that the user has given little feedback on. As mentioned in the Introduction, adopting the collaborative filtering approach poses several novel challenges, because our data items *overlap* due to their tree-structured nature. We describe a solution to these issues in Section 4.3.

4.2 Constraint-based Collaborative Filtering

In this section, we consider an approach to collaborative learning based on MIRA. For each user, we can augment his or her feedback (expressed as *linear constraints* fed into the MIRA algorithm) with feedback from others. From this we will learn a more general weight assignment for the user. The benefit of this approach is that the basic setup of the learning problem remains essentially the same, making the strategy likely to have high effectiveness in learning scores.

To start, we can consider how to combine the constraints expressed by *all* users, and to find an optimal assignment of weight values. This idea encounters difficulties when there is inconsistency among different users' feedback. We can resolve this with the following intuition. For each given user u, feedback from other users should have different degrees of influence, based on how consistent their standards are with those of u. Feedback from "similar" users sharing consistent standards should be weighted more than feedback from highly incompatible users.

This requires that we develop a measure of how compatible one user's feedback is with that of another user. For a given user u_1, we take each unit of feedback (linear constraint) from some *other* user u_2 and assign it a "penalty". This penalty indicates the significance of violation of the constraint, and is based on a similarity measure between u_1 and u_2's overall query and feedback patterns.

We use such penalty values (Section 4.2.1) to adapt the MIRA algorithm to directly encode the penalty of each constraint in its loss function (Section 4.2.2). Since MIRA is an online approximation approach with provable upper bounded cumulative loss, this adapted version will overall reduce cumulative weighted loss in an online setting. In the remainder of this section, we explain the details and describe how to combine the approaches in our **Q** System.

4.2.1 Weighting Constraints by User Similarity

As described above, given constraints from the full user community, it will often be impossible to find a set of compatible feature weights. In contrast to "hard" constraints as described above, *weighted* constraints [64] allow us to violate some requirements, with some penalty. In our setting, the more "similar" two users are, the more impact each user's feedback should have on the other's. We can map this problem of computing edge costs for a given user as a weighted constraint satisfaction problem as follows.

For a given user u_1, consider each linear constraint C provided by any user u_2. Associate this constraint with a "penalty" or weight — indicating its significance of violation — obtained from similarity measure between u_1 and u_2:

$$\text{Pen}^{u_1}(C) = \text{Sim}(u_1, u_2), \tag{4.1}$$

where $\text{Sim}(u_1, u_2)$ is a similarity measure between two users. In our **Q** System, we use cosine similarity between two users' weight vectors, except that for each user's own constraints, their penalty values are taken to be infinite (a hard constraint) rather than 1.

4.2.2 Weighting the MIRA Loss Function

One naive method to find optimal weight assignments for all users, given a network of weighted constraints, is to aggressively minimize the total weight of constraints that are violated. However, this approach has several shortcomings. First, the weighted constraint satisfaction problem is NP-hard in the number of constraints. When a large amount of joint feedback is available, the computation becomes intractable. Second, solutions may evolve dramatically when constraints are added. There is no guarantee that the changes to solutions are indeed beneficial. By contrast, the original MIRA algorithm formulation in Algorithm 1 enables "incremental" (thus efficient) weight updates that ensures stability: each new solution needs to be close to its previous version. It also has provable bound on cumulative loss incurred [20].

We wish to use this notion of weighted constraint but still preserve the benefits of MIRA. To achieve this, we directly incorporate the constraint penalty into the loss function. Given a linear constraint C involving two trees T and T', we reformulate the loss function for user u as follows

$$L_u(T, T') = \mathrm{Pen}^u(C)(|E(T)\backslash E(T')| + |E(T')\backslash E(T)|), \tag{4.2}$$

where $\mathrm{Pen}^u(C)$ denotes the penalty value of a constraint as defined in Equation 4.1. The user's own constraint receives penalty value 1, which is consistent with the original loss function of Section 2.3.3.

The benefit of the loss function is clear once we see the formula for incremental updating weight vector in MIRA. In fact,

$$\overrightarrow{\mathbf{w^{t+1}}} = \overrightarrow{\mathbf{w^{t}}} + L_u \times \overrightarrow{v}, \tag{4.3}$$

where the new weight vector is updated by adding a "delta" vector $\overrightarrow{v}$ (obtained from feedback) multiplied by a scalar. The length of this "delta" vector is proportional to the updated loss function L_u in Equation 4.2. Clearly, large constraint penalty forces the learning algorithm to aggressively update the weight vector.

We outline below how we incorporate this method in the **Q** System.

1. The online learner (MIRA) is triggered as usual: it is applied immediately whenever feedback is received. The user's own weight vector will be updated with new feature

weights, as the online learner updates these for the given user.

2. Whenever any user's feedback is processed, the feedback is propagated to all other users in the community.

3. Upon receiving such feedback originating from some other user, the current user will perform online learning as if the feedback originates locally, but will use the weighted loss function in Equation 4.2.

This scheme has a running-time advantage over the naive weighted constraint satisfaction problem. It also has the benefit that it is directly incorporated in each learning step.

4.3 Collaborative Filtering-based Learning

An alternative approach to personalized ranking is to look at combining not the *constraints* across users, but rather their different *feature weights*. For any particular user, we may only have a small set of existing weights available (learned from small amounts feedback), but we wish to *predict* all feature weights for the user, given the feature weights from the entire community of users.

This motivation of uncovering missing scores from a set of existing weights, is very similar to that of collaborative filtering [73], which has been very successful in a variety of Web contexts, such as movie recommendation on `Netflix`. In this section, we give background on collaborative filtering and explain why and how we must adapt the techniques to our structured setting. One key observation is that in our online setting, we would like each weight update to have a *stability* property, such that each successive feature weight vector remains relatively "close" to the previous solution.

4.3.1 Background: Recommendation Systems

The goal of a recommendation system [51] is to predict how likely a user will prefer an item, based on past history. In our setting, an "item" corresponds to the feature weights for each user's search graph. Recommendation systems are typically classified into three main categories: content-based methods, collaborative filtering methods, and hybrid methods.

Content-based methods typically recommend items to a user based on what kind of items the user likes the most, by computing item similarities and examining the user's past history. In some cases they may alternatively model user profiles to discover similar users. Content-based recommendation methods are often limited, since they usually consider only the history of the particular user, largely ignoring correlations within the huge set of users. To address this shortcoming, recent methods mostly build on collaborative models [73], which aim to leverage ratings from other users as well.

Collaborative filtering (CF) methods are further divided into **neighborhood-based** and **model-based** approaches. A neighborhood-based (or memory-based) method usually predicts a rating based on an aggregated score derived from the entire or a subset of the ratings (for example, ratings from similar users). In contrast to neighborhood-based methods, model-based methods seek to build effective statistical models that best reflect the rating behaviors and patterns. These models are trained from existing data and are be applied to predict unknown ratings. Examples of such models include Bayesian networks, clustering, regression, and Singular Value Decomposition (SVD). Recent work has shown that model-based CF methods give better overall predictions [73], provide intuitive reasons for recommendation, and better addresses issues such as sparsity (few existing ratings available) and scalability.

Finally, **hybrid approaches** exist that incorporate both content-based and collaborative methods [73]. Such techniques seek to exploit both semantic knowledge of the data, plus the ability to find structure among the features.

Model-based CF as a building block. Among the collaborative filtering methods mentioned above, model-based CF best addresses our domain: (1) we assume limited domain or content knowledge, yet seek to predict missing feature weights based on existing feature weights learned for each user from feedback so far (we have sparsity of feedback and limited ability to model content); (2) we wish the collaborative prediction to be fast so that it can be seamlessly combined with each user's online learning (we desire high scalability).

Collaborative filtering is connected to our problem setting in the following sense. CF usually takes as input a set of N users and a set of M items. In our setting, items correspond to features since they compose costs of edges and nodes in user's search graph. Each item

j receives a rating R_{ij} from user i. The value R_{ij} can be a real number or unknown. In our setting, item rating corresponds to feature weight. We view R as a matrix where rows and columns correspond to users and items, respectively, and entries correspond to rating values. The goal of collaborative filtering is to uncover the unknown ratings from the matrix, so that the predicted ratings best reflect the user's underlying preference.

We choose to apply collaborative filtering on feature weights, instead of on the set of Steiner trees. The reason is that there are exponentially number of such trees in the entire search graph but linearly many features. Given that trees are already decomposed into features, it is more effective to perform collaborative filtering on them.

4.3.2 Choice of Model-based Approach

There is already a large body of existing collaborative filtering techniques. For our purpose, we choose to adapt *latent semantic CF models* [42], which introduce latent class variables to describe factors that explain prediction. In our setting, these latent factors correspond to users' underlying standards: Suppose that we know there are K co-existing standards (or "world views"). Also suppose that we know each user's world view, and under each world view which features are "good". Then we can easily compose such knowledge together and derive personalized standards. More precisely, if we let $U^T_{ik} = 1$ if user i adopts the k^{th} view and $U^T_{ik} = 0$ otherwise, and let $V_{kj} = 1$ if feature j is "good" under view k and $V_{kj} = 0$ otherwise, then the (i, j) entry in $R = U^T V$ will precisely describe if user i thinks feature j is "good".

In fact, we are *decomposing* individual weights using each individual user's view (the latent variable that explains predictions). The challenge here is that we do not know these latent values. We must learn them from a small amount of existing information on the feature weights. We explain below how to adapt a latent variable-based approach to solve the problem.

4.3.3 Matrix Factorization-Based Approach

Even within the space of latent semantic CF models, there are many options. Our choice of algorithms was governed by the concern that in the **Q** System, features are associated with every edge and node in the search graph. Among the many potentially relevant CF methods, *matrix factorization methods* [60] have been shown to have excellent overall prediction performance and scalability. We thus adapt these methods to perform prediction.

Matrix factorization methods use dimensionality reduction to construct latent variables for both users and items. Let $U \in \mathbb{R}^{D\times N}$ be the latent user matrix, where the column vector u_i consists of all D latent factors for user i. Similarly, define $V \in \mathbb{R}^{D\times M}$ to be the latent item matrix. The key assumption in a factor model is that, given U and V, the rating matrix R is generated according via the following computation:

$$\Pr\left[R_{ij} = r \mid u_i, v_j\right] \sim \mathcal{N}\left(r \mid u_i^T v_j, \sigma^2\right). \tag{4.4}$$

After the incomplete ratings matrix R is computed, the next step is to perform *model fitting*, which finds values of U and V that are "optimal," i.e., maximize the likelihood of (U, V) given R. Unfortunately, this approach will typically overfit the model, since there are often only a small number of training examples observed. The Probabilistic Matrix Factorization (PMF) [60] model addresses this issue.

Model description. PMF is a technique proposed to enhance Singular Value Decomposition (SVD) and to address the issue of sparse data. The key idea of PMF is to specify a zero-mean prior of latent factors U, V to avoid overfitting.

$$\Pr\left[U\right] \sim \prod_{i=1}^{N} \mathcal{N}\left(u_i \mid 0, \sigma_u^2 I_N\right), \quad \Pr\left[V\right] \sim \prod_{j=1}^{M} \mathcal{N}\left(v_j \mid 0, \sigma_v^2 I_M\right) \tag{4.5}$$

where I_N is the identity matrix of size $N \times N$ (same for I_M). We sketch the corresponding graphical model in Fig 21.

Model fitting. The task of model fitting is to find best parameters for the model, given some available input data. Fitting the above model requires to maximize the the log-likelihood of the posterior distribution of parameters U and V, given some ratings (but

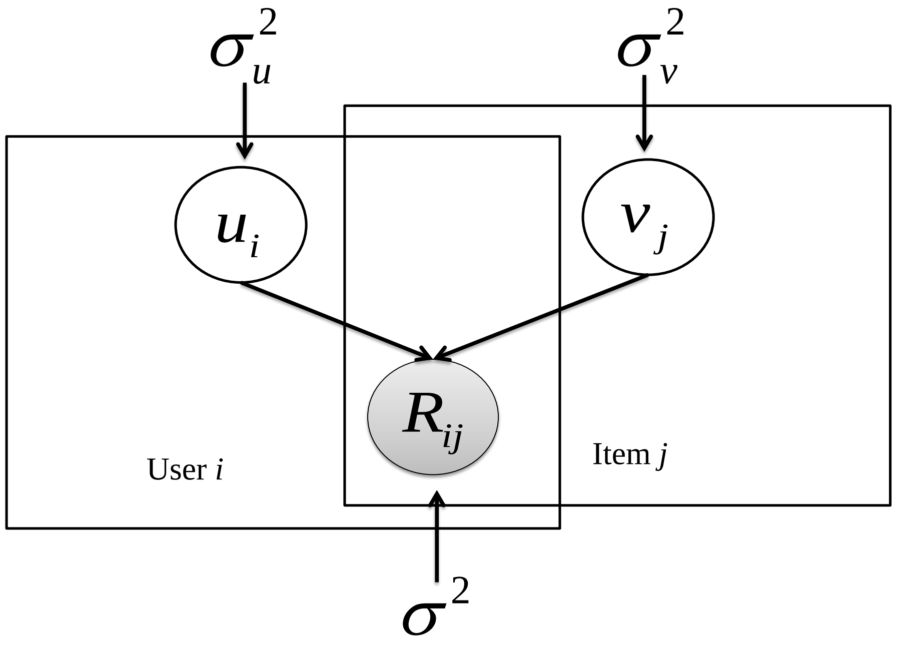


Figure 21: Graphical model for probabilistic matrix factorization (PMF)

usually not all) in R, as follows:

$$\begin{aligned}
&\log \Pr\left[U, V \mid R, \sigma^2, \sigma_u^2, \sigma_v^2\right] \\
=&\log \Pr\left[R \mid U, V, \sigma, \sigma_u^2, \sigma_v^2\right] + \log \Pr\left[U \mid \sigma^2, \sigma_u^2, \sigma_v^2\right] + \log \Pr\left[V \mid \sigma^2, \sigma_u^2, \sigma_v^2\right] - \log \Pr\left[R \mid \sigma^2, \sigma_u^2, \sigma_v^2\right] \\
=&-\frac{1}{2\sigma^2}\sum_{i=1}^{N}\sum_{j=1}^{M}\mathbf{1}_{ij}\left(R_{ij} - u_i^T v_j\right)^2 - \frac{1}{2\sigma_u^2}\sum_{i=1}^{N} u_i^T u_i - \frac{1}{2\sigma_v^2}\sum_{j=1}^{N} v_j^T v_j \\
&-\frac{1}{2}\left(\left(\sum_{i=1}^{N}\sum_{j=1}^{M}\mathbf{1}_{ij}\right)\log\sigma^2 + ND\log\sigma_u^2 + MD\log\sigma_v^2\right) + C,
\end{aligned} \tag{4.6}$$

where $\mathbf{1}_{ij}$ is the indicator function specifying if user i has rated item j and C is a constant. In our setting, user i has rated item j if her feedback has affected feature j. Maximizing the above equation is equivalent to minimizing the summation of squared errors according to the following objective function (by taking partial derivative with respect to U, V)

$$\begin{aligned}
F(U, V) = &\sum_{i=1}^{N}\sum_{j=1}^{M}\mathbf{1}_{ij}(R_{ij} - u_i^T v_j)^2 \\
&+ \lambda_u \sum_i ||u_i||^2 + \lambda_v \sum_j ||v_j||^2,
\end{aligned} \tag{4.7}$$

where $\lambda_u = \sigma^2/\sigma_u^2$ and $\lambda_v = \sigma^2/\sigma_v^2$ are regularization coefficients. The least sum of squared errors-based approach provides an alternative explanation of incorporating Gaussian priors, which is to avoid overfitting through regularization. Learning U and V can be achieved using gradient descent.

Adapting PMF to the Q System. To implement the above factorization techniques within a search-driven integration framework, the **Q** System creates, for each user i, a feature weight vector and variables $\{\mathbf{1}_{ij} | 1 \leq j \leq M\}$, indicating which features have received feedback from user i. During factorization, only features with $\mathbf{1}_{ij} = 1$ have their weights stored in R; features with $\mathbf{1}_{ij} = 0$ have rating $\perp$ (unknown). Then R is factored into, and updated, to the product of U^T and V. Finally each user i receives vector R_i as updated feature weights. This component is periodically invoked.

4.3.4 Stabilizing through Regularization

The above method directly adopts matrix factorization, where users and features are "clustered" to different views. However, this approach does not ensure the stability property mentioned at the beginning of this section. In normal matrix factorization, the adjusted weights may deviate significantly from their previous values, and such changes may not be natural to the user (resulting in wild fluctuations of query answer rankings), nor beneficial. To improve stability (a key feature offered by the MIRA algorithm), we add into the objective function in Equation 4.7 a new regularization term which measures the (L2-)distance between adjusted weights and old weights, as follows:

$$F(U,V) = \sum_{i=1}^{N}\sum_{j=1}^{M} \mathbf{1}_{ij}(R_{ij} - u_i^T v_j)^2 + \lambda_u \sum_i ||u_i||^2 + \lambda_v \sum_j ||v_j||^2 + \lambda ||V - V^{\text{old}}||^2. \quad (4.8)$$

Just as with PMF, we can apply gradient descent techniques to learn U and V.

4.4 Experimental Analysis and User Study

In this section we present our experimental evaluation of the proposed methods for *combining and resolving* users' feedback in the **Q** System. We seek to validate that our collaborative learning techniques improve the system's ability to accurately predict a user's assessment of the viability of schema mapping edges (and thus query answers that use these mappings), even when different users have different value-assessments ("world views"). We also validate that our implementation incurs reasonable overhead beyond that of learning in the existing framework, and we seek to better understand the differences among the different collaborative learning strategies for combining results. To understand performance relative to different factors, we first conduct a set of **synthetic experiments**, showing that the system can learn the mutually incompatible preferences of different sets of users rapidly. Later, we use our neuroscience data portal [55] to conduct a **real user study** over actual neuroscience data.

4.4.1 Experimental Methodology

Conducting keyword search experiments across integrated data sources poses challenges in (1) recruiting users with representative queries, and (2) identifying a complete set of correct ("gold") scores for answers. Thus we combine synthetic experiments (for diversity) with real user studies (for depth and realism).

Synthetic experiments. To create our first set of experiments, we consulted users (with knowledge across three domains, bioinformatics, movies, and geography) to identify different classes of sources and links that might be preferred by different subcommunities. Our experiments then simulate multiple users' feedback, to see how quickly we can "recover" the users' rankings. In each iteration, the **Q** system returns the top-k ranked Steiner trees for the keyword query. Then we simulate the user's feedback: a Steiner tree receives **positive feedback** if all of its edges are viewed as correct alignments by the current user, and **negative feedback** otherwise. Next we move "round-robin-style" to the next query and repeat the process. Note that our definition of a correct Steiner tree is very strict, as there might be additional useful alignments that are not specified by the user or in the schema. The **Q** system uses the feedback to learn adjustments to the feature weights and updates costs of edges in the search graph for the current user. The system then propagates the feedback to other users using one of the collaborative learning methods. We describe further details in Section 4.4.2.

User studies. While we could not recruit users across many diverse domains, we were able to use the IEEG.org neuroscience data portal [55] to conduct a user study, using actual neuroscience data and queries. We describe the setup in Section 4.4.5.

Implementation. Experiments were conducted using our implementation of the **Q** System, which comprises approximately 60k lines of Java code. Evaluation was done on an Intel Xeon CPU (2.83GHz, 2 processors) Windows Server 2008 (64-bit) machine with 24GB RAM, using Java SE 1.60.11 (64-bit). For each dataset, the **Q** System first loads its schema (without knowing foreign keys and/or personalized standards) and constructs the schema graph. We run in parallel a set of schema matching primitives from the COMA++ schema matcher (Community Edition) to compute a similarity score between [0; 1] for every alignment feature. We prune potential edges for which all matching primitives give low similarity scores.

Parameter Settings. We use $k = 10$ as the number of top queries to compute answers for each keyword search. We use the following COMA++ schema matching primitives: data type similarity, string edit-distance, string q-gram distance, semantic similarity, and instance matchers. Initial matcher weights are trained offline before the experiment using a very small set of example attribute pairs. During collaborative filtering, we use cosine similarities on feature vectors to measure user similarity. We use $D = 1$ as the number of latent variables for experiments with consistent feedback and use $D = 2$ for datasets that permit multiple standards. We set the regularization strength to be 0.1.

Roadmap. We consider the following questions:

- Do collaborative learning methods help "bootstrap" the system's ability to predict which *edge* alignments are accepted by a given user as correct ("gold"), vs. invalid? (Section 4.4.2.)

- How does the learning process over edge costs translate into improvements in overall query answer quality? (Section 4.4.3.)

- How much running-time overhead is added to the existing learning algorithms in the **Q** System? (Section 4.4.4.)

- Do the results from simulated users across multiple domains carry over to real users in neuroscience? (Section 4.4.5.)

There are two independent parameters we seek to measure (and minimize). The first is the number of rounds of learning required to achieve correctness, according to the user's preference; we term this the *learning rate*. The second is the *running time* as we scale the number of feedback samples.

4.4.2 Learning Rate: Edge Quality

The strongest correctness criteria for the **Q** System is for it to correctly distinguish between what the user would define as correct ("gold") versus invalid ("bad") **edges** in the graph — i.e., schema alignments (across metadata nodes) or record links (across data nodes). If all of the edges are correctly identified, then any query returned by the system should also be correct. To achieve this, in general the system must receive feedback on every feature (and, since each edge has at least one unique feature, each edge) to appropriately adjust its weight.

4.4.2.1 Users, Queries, and "World Views"

For clarity of results, all of our experiments focus on the ability of the system to identify (and return in the top-k answers) a set of *correct* results as viewed by each user; and to distinguish these from *incorrect* results according to the user — i.e., for each user we seek to classify potential answers as good or bad.

For each dataset, we simulated 10 users posing keyword queries in a round-robin manner. Our query workload consists of 10 queries for each dataset, each of which has low selectivity. During each step, each simulated user randomly picks a query from the workload.

To model users with different "world views" and determine if the **Q** System effectively combines and resolves feedback, we use a "graph region-based" model, in which each user has a "focused" sub-region of "expertise" within the search graph, and a world view (of what links are good) in the region. Each simulated user can:

- within her expertise area, *update* "good" edges' feature weights to have costs lower than those of "bad" edges,
- outside of her expertise area, *query* and get answers with consensus feature weights.

We assign a single consistent world view to each basic dataset (Bioinformatics, IMDB and Mondial), but partition each into different "regions of expertise" and map the regions to different user groups. Here collaborative learning will help users find quality answers outside their region of expertise.

To look at collaborative filtering among *different* world views, our **IMDB+DBpedia** dataset takes a set of ambiguous record links across the tables, and assigns each user a different view of which links are correct. Finally, we create **Bio2** using the same data as **Bio**, but allows users to specify if they trust the record links (such as table **interpro_entry2pub**).

4.4.2.2 Learning rate with consistent feedback

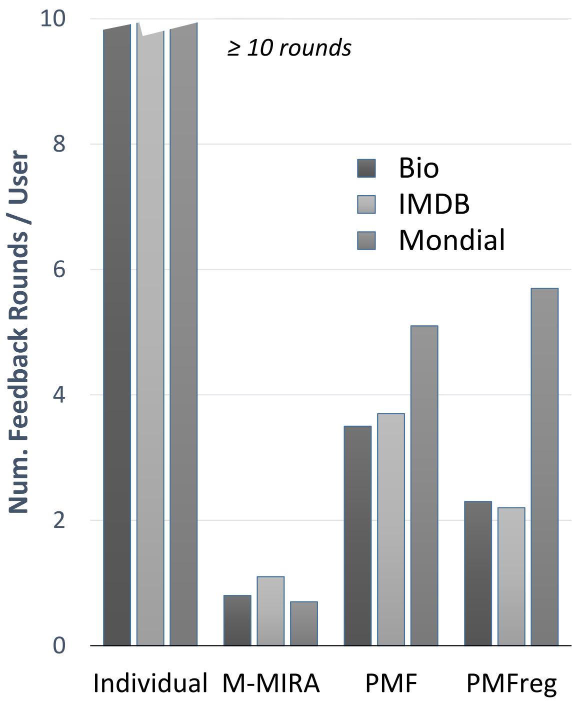


Figure 22: Learning rate: feedback rounds to separate costs of gold vs. bad edges.

We start by measuring how quickly (in terms of amount of feedback required) our proposed collaborative learning methods can help a given user achieve the edge-separation

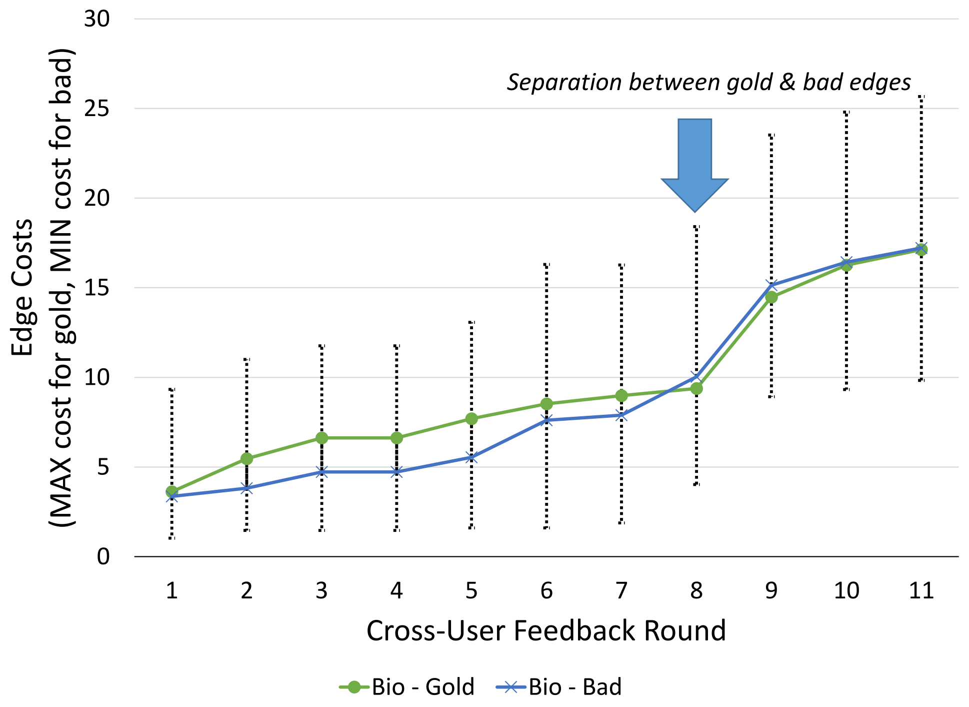


Figure 23: Costs of gold & bad edges after each feedback round, for **Co-MiRA** over **Bio**. Distribution of edge costs is indicated with error bars.

property. Figure 22 shows that if each user learns a ranking function in isolation (**Individual** strategy), as in [77], it takes more than 10 rounds per user. The collaborative MIRA algorithm of Section 4.2, **Co-MiRA**, achieves much faster amortized results by combining user similarity and multiple users' feedback into the loss function. Its performance is essentially indistinguishable from running MIRA across the set of users globally (not shown).

The collaborative filtering algorithms proposed in Sections 4.3.3 and 4.3.4 are denoted **FF** (feature factorization through probabilistic matrix factorization) and **FF-reg** (feature factorization with regularization). These show significant gains over the **Individual** learning baseline. However, they have much more moderate influence on each user's feature weights than full-fledged learning-from-combined-feedback methods. Thus, under the consistent world-view assumption, **Co-MiRA** is clearly the most effective approach.

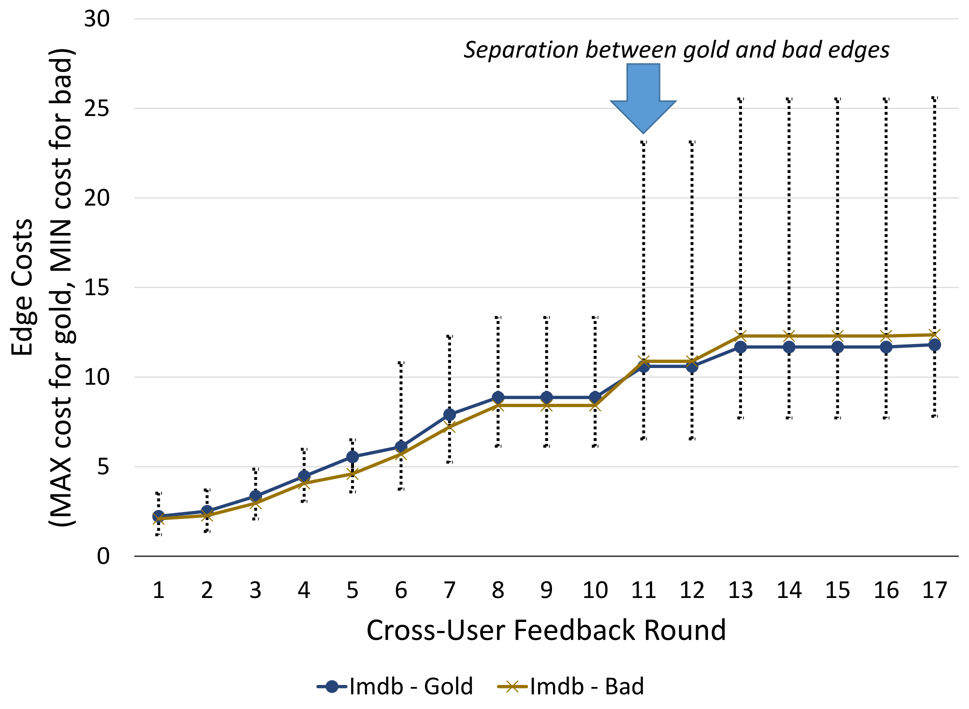


Figure 24: Costs of gold & bad edges after each feedback round, running **Co-MiRA** on **IMDB**. Distribution of edge costs is shown via error bars.

4.4.2.3 Cross-user learning with consistent feedback

We next look in more detail at what happens as feedback is provided across users, for **Co-MiRA** since it performed best. First, Table 1 shows that in each round, every user is receiving substantial feedback on edge weights from other users' feedback.

Dataset	Avg no. edges receiving feedback per round
Bio	3.38
IMDB	6.0
Mondial	8.29

Table 1: Average number of edges receiving feedback per round, until full separation is achieved.

We can see the effect by looking at round-by-round changes for **Co-MIRA** on the **Bio** dataset. We pick one user and plot (in Figure 23), as each distinct user provides feedback in round-robin fashion, the **maximum-cost gold edge** in her graph, and the **minimum-cost bad edge**. We additionally use error bars to show the range of values for the bad edges (bars going upwards from the minimum-cost bad edge) and gold edges (bars going downwards from the maximum-cost gold edge). As we see in the figure, after 8 users' feedback, the values completely separate.

Figure 24 shows a very similar pattern, for the **IMDB** dataset. Here, 11 round-robin steps are required, meaning that we cycled back to the first user. We observe that although **IMDB** has fewer edges than **Bio**, and receives more edge feedback per round, the diversity of the queries (hence the machine learning algorithm's ability to learn to allocate correct weights across the edges) is lower.

4.4.2.4 Learning rate with conflicting feedback

The previous experiments focused on settings where a single set of consensus weights is possible. In general, we target settings where data is *inconsistent*, hence techniques based on trying to find global assignments would not converge.

Here we use the two partly-synthetic datasets mentioned in Section 4.4.2.1. The joint dataset **IMDB+DBpedia** includes entities for people. We created multiple record linkage tables (choosing one alignment out of every ambiguous matching set) for these entities, respectively, using string similarity measures to connect records in IMDB to those in DBpedia. Each user receives a "trusted" record linkage table. We similarly generated **Bio2** by allowing users to specify which record linkage tables they trust.

Each record linkage table represents a world-view. We expect the system to learn lower costs for each user's preferred record linkage edges (if the user has a preference), when compared to the edges from others tables. Of course, for edges that have consistent global views, we expect our learning algorithm to correctly separating their costs from costs of invalid edges. We measure the diversity in the record links in Table 2.

For this experiment, we slightly relax the separation requirements due to scale: with so many edges, for a given query workload not every edge receives significant feedback. We focus on **95% separation**, i.e., 95% confidence interval of costs for gold edges (mean

Name	**Bio2**	**IMDB+ DBpedia**
Avg. conflicting edges (per user pair)	2.84	2.13
Max. conflicting edges (among all pairs)	8	4

Table 2: Number of conflicting edges in datasets allowing multiple views.

±1.96 sd) is completely disjoint from 95% confidence interval of costs for bad edges. We consider two baseline strategies for combining inconsistent feedback: **Individual**, which learns separate weights for each user; and **Average**, which combines weights from different users via a standard averaging function.

Dataset	**Indiv.**	**Avg.**	**Co-MiRA**	**FF**	**FFreg**
Bio2	10+	10+	6.1	**5.2**	**5.2**
IMDB+ DBpedia	10+	10+	7.8	7.1	**7**

Table 3: Learning rate: minimum number of feedback rounds to achieve separation of costs of gold edges and invalid edges.

As we see in Table 3, neither strategy achieves separation in under 10 rounds per user. In this case, the two collaborative filtering methods based on matrix factorization, **FF** and **FFreg**, achieve the best results. The **Co-MiRA** algorithm produces benefits, but under this setting with conflicting views, it is less effective.

4.4.3 Learning Rate: Query Answer Quality

The previous experiments showed the system's ability to distinguish between correct and invalid *edges*, independent of their impact on query results. We now consider their impact in a more workload-aware fashion, i.e., how quickly the separation of good and bad edges leads to better query answers. To do this, in addition to the set of training queries, we also supply for each dataset a set of 5 holdout test queries. In each round of per-user feedback, we check if the holdout queries are correctly answered.

Results (for the globally consistent setting, then the multiple-world-view setting) are presented in Table 4. We observe that here, **Co-MiRA** again works best under a single globally consistent world-view. The three methods perform similarly for multiple world-views, but in fact **Co-MiRA** has a slight edge for the **Bio2** setting.

Dataset	Baseline	Co-MiRA	FF	FFreg
Globally consistent				
Bio	10+	**0.8**	2.3	1.4
IMDB	10+	**1**	3.7	2.2
Mondial	10+	**0.7**	5	5
Multiple world-views				
Bio2	10+	**3.8**	5	5
IMDB + DBpedia	10+	**7**	**7**	**7**

Table 4: Minimum number of feedback rounds to achieve full precision & recall of answers to holdout queries.

4.4.4 Running Time vs. Scale

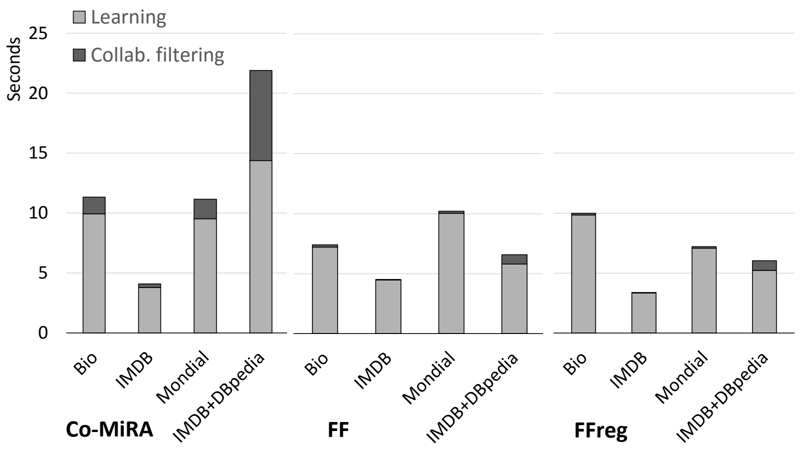


Figure 25: Average time spent on collaborative filtering versus overall query answering & learning, in each feedback round.

Our setting requires interactive, *online learning*, hence a key consideration for our collaborative learning techniques is not only answer quality, but also running-time overhead. Figure 25 plots two components: the time spent in each feedback round on query answering and online learning (the lower, light-gray bar), and then the dark bar shows the additional overhead added by the different collaborative filtering techniques. The numbers will differ somewhat across the methods because (1) exact features being updated (and hence total number of them) are different during each round and (2) the time includes that of computing top-k query answers (Steiner trees), which differ when the underlying edge costs change. We observe that **Co-MiRA** adds minor overhead for the smaller datasets, but as we scale up to the **IMDB+DBpedia** dataset, it adds a further 50% overhead to the response times of the system. In contrast, the collaborative filtering algorithms add very minor overhead.

Discussion. All three of the methods proposed in this chapter appreciably improve the **Q** System's ability to learn from inconsistent community members' feedback. **Co-MiRA** is more effective for relatively small domains where the diversity is not extreme. However, **FFreg** (which dominates **FF**, thanks to its additional regularization step) is significantly more efficient, and it is the preferred choice at scale or with high diversity.

4.4.5 User Studies

As a final validation of the effectiveness of our techniques, we conduct a brief user study over neuroscience data. For this user study, our goal is primarily to show that collaborative learning can effectively set the ranking function for a user, such that they will frequently not have to give additional feedback to the system to get their desired answers in the correct order.

Dataset. The user study was conducted using the data from the IEEG.org neuroscience data portal operated by the authors, with over 2000 datasets and 1000 users [55]. These datasets come from approximately 40 different sources, and they vary in several ways. Data was often collected for different target subfields (e.g., epilepsy research vs. behavioral research); on different organisms (e.g., humans vs. rats vs. dogs); from different institutions or labs. Some of the data was "raw" data and other datasets contained annotations, created by different classification and detection algorithms, "overlaid" on related datasets. This diversity enabled us to construct a fairly large space of features over which users could (implicitly through feedback) express preferences.

Users and tasks. We conducted our user study with 10 users of varying backgrounds including computer science, bioengineering, and neuroscience. Each user was given a brief description of the kinds of data available, and to come up with a preference for certain kinds or sources of data, as well as an anti-preference for other kinds of data (we did not tell what users needed to choose). Then the user was asked to pose a series of (10 different) keyword queries, and, if the results were not according to preference, to provide feedback on a small number of results to identify the desired (and undesired) data. We provide an excerpt of some of the queries in Table 5. For each of them, we also list the number of results, the approximate number of results classes and example of classes that users have used to rank; however, the queries were not pre-specified and some users posed highly diverse searches. The average query returned between 50 and 100 results, although this varied greatly by the search terms.

Query	**Results**	**Result classes**	**Example preference features**
"Seizure detection"	56	8	Detection algorithms; organizations
"Mayo"	70	3	Data collection vs channel
"Dog"	24	3	Organization
"Female"	28	4	Organization
"Hospital"	569	6	Animal vs human; organization
"Seizure"	34	8	Detection algorithms; organization
"Rat"	842	20	Organization
"EEG"	1530	15	Organization
"University"	545	10	Animal vs human; organization
"Children seizure"	59	8	Detection algorithm; organization

Table 5: Excerpt from queries used in user study

We sought to answer two main questions:

1. Can the **Q** System quickly learn to differentiate the preferences for graph *nodes* based on organism type (e.g., human vs animal), source, etc?

2. Can it quickly learn to differentiate user preferences for *edges* (classification tools that produce annotations)?

Methodology and results. To limit the time each user needed to spend, we focused our study on the best-performing algorithms, **Co-MiRA** and **FFreg**. For each user, we measured, out of their 10 rounds of queries, (1) for how many queries they needed to provide (any) feedback; (2) for how many *tuples* they needed to provide feedback; (3) how frequently they were immediately satisfied with the ranking of the returned answers. Each user had preferences based on the data type and source (node preferences) as well as based on annotation or classification relationships produced by detection tools (edge preferences). Results are shown in Table 6.

Method	**Co-MiRA**	**FFreg**
Avg no. queries that need feedback	2.4	3.2
Avg no. tuples/query given feedback	1.1	1.2
Avg no. queries showing "satisfactory" results without feedback	7.6	6.8

Table 6: User study results

Overall, out of 10 queries, on average across the user population, only 2.4 (for Co-MiRA) or 3.2 (for FFreg) queries required any feedback whatsoever; i.e., for the majority of the queries, the system was able to provide the correct ranking. Even when feedback was required, on average it was 1-2 tuples' worth of feedback. Thus we conclude that indeed the collaborative learning techniques greatly reduce the overall amount of feedback required to satisfy the users' information needs.

4.5 Conclusion

In this chapter, we extended the notion of *search-driven data integration* to learn a custom ranking function for each user in the system, based on collaborative learning techniques that adapt to the user's preferences, query patterns, and similarity of feedback to other users. We implemented and experimentally validated several approaches to this problem, and found that probabilistic matrix factorization-based collaborative filtering techniques, in particular, seem to have good characteristics for both running time and learning rate. The resulting system is capable of tailoring its behavior when there are multiple communities with different information needs or preferences, as we validate with a user study.

Chapter 5

Interactive Query Debugging in the Q System

Search-driven integration systems usually respond to user's information need, expressed as keywords, by returning a set of top-k answers. These answers are usually relevant when integration quality is good. However, this is not always the case. For a given keyword query, it is entirely possible that users may not see correct (hence satisfactory) results due to errors in keyword matching, link prediction or result ranking. For instance, problematic *name disambiguation* may match query terms to wrong nodes in the input graph, generating incorrect results. In addition, even when keyword terms are matched correctly to appropriate terminal nodes, paths or trees connecting them may be wrong when they use problematic links. Finally, expected items may appear with *low ranking*.

Incorrect results may also appear in what are commonly termed *knowledge graphs* that may arise in searchable, enterprise-wide "data lakes," Linked Open Data, Web knowledge extraction (e.g., Probase, Nell, Google's Knowledge Vault, DBpedia, YAGO), or in crowd-sourced integration settings. These data resources leverage both human-curated and automatically extracted or linked data, typically in a graph representation where nodes represent data and weighted edges represent confidence values. Keyword search-over-graphs can be used to answer user queries about relationships over this data; and machine learning can be used to learn which data components are of interest, by taking user feedback over returned top-k answers. However, due to errors made by crowd workers or (semi-)automatic extraction algorithms, knowledge graphs can have so much "noise" that certain top-k queries *only* return useless answers.

Unfortunately, current search-driven integration paradigm does not address the issue of returning only incorrect results, for both architectural and technical reasons. The current architecture, outlined in Section 2.2, lacks a component that, when no correct results occur, triggers an interactive procedure to help user find relevant query answers. Even worse, technically, online learning algorithms for learning to rank from query result feedback, such as MIRA, fail to work in this case: MIRA requires *at least one positive example* and *at least one negative example* to form the set of linear constraints, which specifies that cost of each good tree is less than cost of each bad tree plus a loss function. Hence, these methods are not applicable when data expert considers all examples to be negative.

For these reasons, we argue that search-driven integration needs an additional *debugging* component to aid user to tackle incorrect answers. A major obstacle in search-driven integration is that the underlying system only learns from feedback on a restricted query results, i.e., results for *user's keyword query*. **This inherently limits the full potential of the huge space of potential samples that the system can learn from.** The purpose of debugging, instead of presenting answers to the keyword query provided by the user, is to proactively select *a series of answers or partial answers* for user to give feedback. In other words, different from prior learning models for search-driven integration, the debugging module has the ability to *select any* samples and request their labels from user. These samples are not necessarily answers to user's keyword query, but rather most informative partial answers that best help improve the graph.

In fact, this approach is inspired by the *membership query synthesis* model in machine learning [53], in which the learner *creates* the sample on demand itself before sending it to an oracle for labeling. This model is similar to but still different from the active learning model. In both cases, a learning algorithm has access to unlabeled data and has the ability to explicitly select the next highly informative sample for an oracle's annotation. Both attempt to learn a good model with significantly smaller number of samples requested. However, membership query synthesis self-generates the sample, whereas in active learning we issue a *query* with some objective function to a pool of unlabeled samples. In search-driven integration, the pool of unlabeled samples is the set of answers for a given keyword query.

Our goal of improving graph from expert feedback is very different from crowdsourcing. Our improvement is biased by the user community's information need. In crowsourcing, specific feedback is usually edge-at-a-time and may or may not intersect with candidate answers to user questions. In the search-driven integration model, specific feedback is tree-at-a-time and is specifically guided by each user's information need. In both cases, we want to guide the feedback towards the most informative results. The place where we have a challenge, however, is how to do this in a way that provides both positive and negative training examples. The problem we are confronting is the one in which we are mis-calibrated enough to not know what is likely to be a positive example.

While this problem is motivated from our own experience in deploying search-driven integration in real applications [47], in fact our techniques also generalize to other semi-supervised settings, in which (1) graph links are initially constructed but potential errors exist; (2) experts incrementally query the graph and indicate if results are correct; (3) iteratively, links are either confirmed or removed due to feedback. These applications include path finding in an extracted taxonomy of concepts, explaining relationship among entities in an automatically constructed knowledge base, and route planning.

Ultimately, the purpose of debugging is to cleanse and improve noisy graphs, e.g., schema search graphs for integration or knowledge graphs. There are at least two classes of approaches for debugging: query-based methods and graph-based methods. A query-based approach aims to find an alternative query that is close to the input query, leaving the underlying graph unchanged, so that the new query can give good answers to the user. Alternatively, graph-based approaches seek to change link status, e.g., (probability of) existence, in the current graph, so that running the same query on the updated graph will yield correct answers. In this chapter, we focus on the second class of approaches. More precisely, the debugging problem we are investigating is the following: **when there are only incorrect answers for a keyword query, how to select a series of partial answers or subgraphs for user's feedback, from which the system updates the graph, until correct answers for the same query are returned.**

In this chapter, we first examine a slightly restricted model of the graph, where each link is classified as either correct or incorrect. We term this model the exact learning model, since we seek to recover exact correctness of all graph links. We start with a worst case, then propose two query debugging methods for this model: a recursive mechanism to *quickly* pinpoint problematic edges and a *greedy* method to prune the space of false answers. We then proceed to an approximate learning setting, in which each link is associated with a probability. There we develop a Bayesian framework to update link probabilities given expert feedback and a current graph snapshot. In addition, to address the problem of limited user attention, we develop prioritization techniques to find the most informative samples for user to label.

To summarize, we make the following contributions in this chapter.

- A formal model description of query debugging in a noisy graph and using expert feedback to learn links, under two different learning models: exact learning and approximate learning;

- A worst case lower bound analysis in the exact learning case where the labeling complexity is large (quadratic), along with efficient recursive search method to identify problematic links;

- A Bayesian update framework to incorporate feedback, taking into account incorrect feedback and user confidence.

- Efficient prioritized search strategies to select candidate results for labeling based on uncertainty reduction.

- Extensive experiments on datasets that span multiple domains.

Roadmap. The rest of the chapter is organized as follows. We formalize in Section 5.1 the debugging problem studied in the chapter. There we also describe two models we focus on, a boolean model where the graph maintains binary classification on edges and a weighted model where edges are associated with probabilities. Next we investigate debugging approaches under these two models in Section 5.2 and Section 5.3, respectively. We present our experimental analysis in Section 5.4 and conclude this chapter in Section 5.5.

5.1 The Debugging Problem

As briefly discussed in the previous section, our focus in this chapter is a graph-based debugging approach, in which the system iteratively learns to adjust the status of underlying graph, so that running the same keyword query over the updated graph produces correct answers. In each iteration, the system asks user to give feedback on a sub-selection of results and executes a learning method to adjust link status accordingly. Therefore, for a given keyword query, our debugging approach consists of two components:

1. **Synthesizer**: this specifies a selection procedure that decides a series of partial results to get feedback on.

2. **Learner**: this includes a learning algorithm that takes the current graph along with user's feedback as input and makes necessary changes to the graph.

More formally, the debugging problem is defined as follows.

Debugging Problem Definition:. Given a keyword query KQ over a graph G returning no correct top-k answers, the debugging problem for KQ is to modify G to G', so that KQ will return satisfactory answers on G'.

In this section, we describe two models that capture "state of the graph" (we discuss selection procedures for these two models in Section 5.2 and in Section 5.3, respectively). These models are also where learning algorithms operate. In general, the goal of learning is to use expert's feedback on result correctness to identify link "quality". Each link is defined by a pair of node and there are $O(n^2)$ of them. We assume that there is a set of true links (the "gold standard"), which we do not know but wish to learn to recover. Given the gold standard, each predicted link is classified as either "correct" or "incorrect", indicating correctness of link prediction. We call this classification the "underlying" label of the predicted link.

To recover the gold standard, we consider two learning settings. The first is an *exact learning* model where links in the graph are classified to either "true" or "false". In this chapter we also use "binary model" to refer to the setting. The second is an *approximate learning* model in which each edge is associated with a probability of being present in the graph, which is useful in deriving confidence and ranking. We also use "weighted model" interchangeably. We formally define the labeling function used in each model below. In both cases, edge labels will be adjusted (through learning) when expert feedback is available and this loop of learning from feedback will continue. The difference between the two models is that the approximate learning model may provide more information but is intuitively hard to learn.

- **Exact Learning:** The labeling function $L : V \times V \rightarrow \{\mathtt{T}, \mathtt{F}\}$ applies to each node pair (i.e., a possible link) and returns a *binary* label where `T` predicts edge existence and `F` predicts the opposite.

- **Approximate Learning:** This is akin to the cost model in Section 2.3. The labeling function $L : V \times V \rightarrow [0..1]$ assigns a predicted probability value to each potential link between two nodes.

Given labels on individual edges, we can then determine the label of a given tree. Here we assume edges are independent of each other, which may not be entirely accurate but is a good model heavily used in keyword search over structured data and crowdsourcing. In the boolean model, the label of a tree T is given by

$$L(T) = \bigwedge_{e \in T} L(e). \tag{5.1}$$

In the weighted model, the probability of tree T is given by the product of edge probabilities, assuming edges are independent,

$$L(T) = \prod_{e \in T} L(T). \tag{5.2}$$

In addition, orthogonal to the learning model, there are also two sources of *external information*, described as follows.

- **Membership Oracle:** This oracle $O : 2^G \rightarrow \{\mathtt{T}, \mathtt{F}\}$ takes a given connected subgraph (usually a tree) as input and returns if the input subgraph is "correct" or "incorrect". We assume a conjunction model where a subgraph is correct iff all its edges are correct. This means that even one false edge results in subgraph classified as "incorrect". Usually a data expert serves as a membership oracle, as is the case in the **Q** System.

Typically membership oracle is expensive to invoke and it is desirable to minimize the number of calls to oracle.

- **Prior Link Prediction:** This information captures the *initial graph state*. Usually the graph does not start from empty. Instead, it may already have some links automatically constructed, even though they might be wrong initially. Sometimes there may even be *predicted probabilities* available, e.g., from link analysis tools. In our **Q** System, links are initially constructed using a library of schema matchers, which assigns every node pair a probability of being linked, which can be used as it is or compared against certain threshold to derive binary link classification.

5.2 Boolean Model

In this section we study the *exact learning* setting, i.e., the boolean model. We first show in Section 5.2.1 that with the absence of *prior link prediction*, even for a single pair of source-destination, the number of calls to the membership oracle is $\Theta(|V|^2)$. In Section 5.2.2, we proceed to show that, when prior link classifiers only make optimistic predictions, querying the membership oracle with a candidate tree either results in confirming the tree (therefore all of its edges) to be correct or leads to strict *progress* where one can *quickly* identify a problematic edge in the tree. This justifies the crucial role of prior information. Finally, in Section 5.2.3, we present a *greedy* approach that attempts to reduce a large amount of false answers, by prioritizing feedback on edges heavily shared across possible answers.

5.2.1 Worst Case Analysis

The worst case graph G_{worst} consists of two cliques connected by a bridge, visualized in Figure 26

Proposition 2 *With no prior link information available, given a pair of node (s, t) in G_{worst}, where s and t lie in two different cliques, the number of calls to identify a path with correct links connecting s and t is $O(|V|^2)$.*

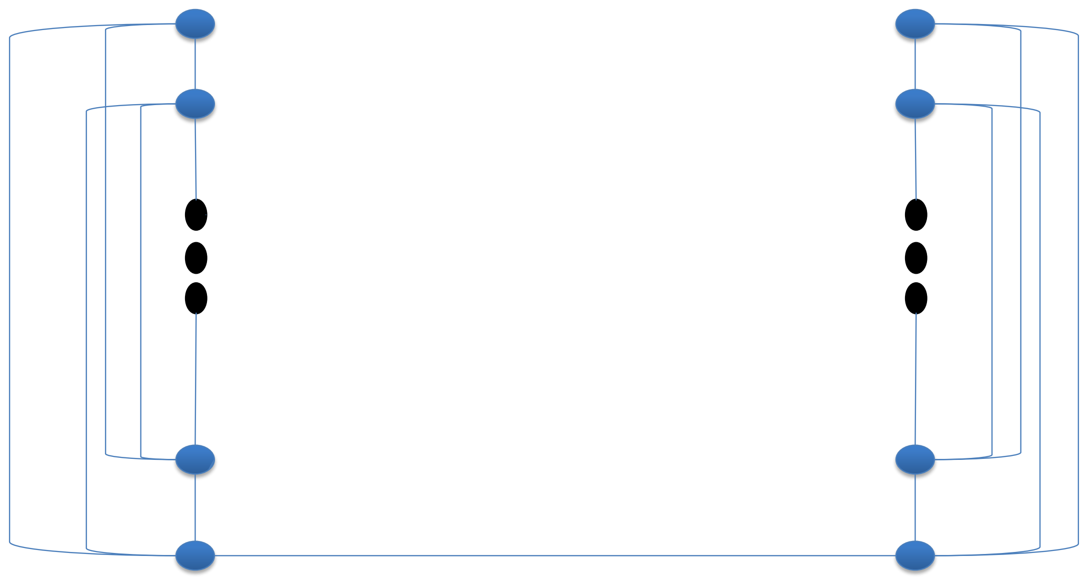


Figure 26: A worst case graph G_{worst} for the boolean model with n nodes. The left part has $n/2$ nodes and the right part has $n/2$ nodes. Either part is a clique. The left part and the right part are connected through a bridge.

5.2.2 Optimistic Priors and Recursive Search

Our above analysis shows that with the absence of prior information, exact learning is almost hopeless even for *one* pair of source destination. Here we show that prior link predictions can indeed be very helpful. Indeed, we show that in the case of optimistic priors, in which the prior always classifies a true link to be "true" but may falsely claim a non-existing link, every call to the membership oracle is making progress in the following sense. If the membership oracle returns yes for an input tree, then we have found a correct tree, hence confirming all its edges to be correct. If not, we can apply a recursive binary search strategy to efficiently identify a false edge, described below.

Proposition 3 *Let G^* be the underlying graph with underlying labeling function and G the* *current graph. Any path in G^* remains in G.*

Algorithm 6 Binary search for a false edge.

Input: Tree T with expert labeled as false
Output: A false edge e

```
1: while $T$ has more than one edge do
2:     $T' \leftarrow$ left half of $T$
3:     $L \leftarrow$ label of $T'$
4:     $T \leftarrow T'$ if $L$ is true, otherwise $T \leftarrow$ right half of $T$.
5: end while
6: return $T$
```

Consider a given path P for which the membership oracle returns "false". We can then invoke the left half of P as input. If the oracle again returns "false" we know that the left half must contain at least one false edge so we recurse. If the oracle returns "yes" instead, we know that the right half must contain at least one false edge so that we recurse there. We repeat this process until the path degenerates to a single edge, which we have identified as a false link and can remove from the graph. In addition, each positive response from the membership oracle confirms a number of edges to be correct. Suppose there are S positive labels during the search, the number of edges that can be classified as true is at least

$$1 + 2 + 4 + \cdots + 2^{S-1} = \Omega(2^S). \tag{5.3}$$

This algorithm is described in Algorithm 6.

Proposition 4 *Whenever the membership oracle returns* `F` *for an input tree T, it takes $O(\lg |T|)$ additional calls to the membership oracle to identify one false edge in T. In addition, if there are S positive labels during the binary search, $\Omega(2^S)$ edges will be confirmed to be true.*

The case for a tree is similar. Instead of continuing with left half, we first compute the centroid of the tree, which divides the tree into several subtrees, each of size at most half of the original tree. We then recursive on these subtrees until we identify a false edge.

5.2.3 Greedily Pruning Multiple Answers

The above method aims to repeatedly find an incorrect link in one of the top trees, thus improving link quality overtime. Its advantage lies in efficiency. Sometimes, for a given keyword query over a large noisy knowledge base, there can be hundreds or thousands of results, with some edges heavily shared among many trees. For instance, our **Q** System returns more than 100 paths for the query "shanghai, China", and edges like "authentic shanghai snack → snack" occur frequently in results. In this setting, the above method may be inadequate since the data expert may not know which result to debug and identifying a false edge with low frequency seems inefficient.

To solve these problems, this alternative method seeks to help data expert clean edges that are relevant to most query results. To achieve this, we first compute, for each link, the number of appearance in the set of returned query results, followed by asking data expert to label links with the highest occurrence. This procedure is repeated for all edges with high frequency, until some correct answer can be found in the updated graph. This algorithm is described in Algorithm 7.

Algorithm 7 Greedily label edges based on frequency.

Input: Set of query trees $A = \{T_i\}$, user U, keyword query KQ, input graph G.
Output: Updated graph.

```
1: Compute for each edge $e$ the count $C(e)$ indicating how many trees in $A$ contains $e$.
2: All edges are initialized to be "unmarked".
3: while $U$ is not exhausted do
4:     Select an unmarked $e'$ to be the edge with the largest $C(e')$
5:     Mark $e'$.
6:     Obtain label of $e'$ and change the graph if necessary.
7:     If recomputing answers for $KQ$ on $G$ yields satisfactory results, stop
8: end while
```

5.3 Generalized Weighted Model

To this point our discussion has focused on learning exact binary labels for links. However, this model is restrictive in many cases. Most importantly, its prediction does not have degree of confidence available. Instead, in this section we consider approximate learning to predict a probability value for each possible link. In addition to providing confidence, these probability values are useful in deriving probability of a subgraph (by assuming edge independence) and in ranking. For example, our **Q** system and other keyword-search-over-structured- data algorithms compute the top-k Steiner trees as candidate answers for a given set of nodes. Moreover, graphs in our targeted applications are usually constructed with some initial probability information: for example, in automatically construction of knowledge bases, links are generated with scoring information [74]. Therefore, it is natural to start from those values and gradually improve over time based on expert's response.

In this model, these probability estimations are dynamically adjusted in a query-driven manner: a data expert indicates if a tree is correct (possibly with some level of confidence), thus updating probability predictions. A key challenge here is how to update probabilities in a belief-centric way. We propose a formal Bayesian approach to update link probabilities in Section 5.3.1. This approach can also incorporates feedback confidence and noisy oracle.

In addition to this Bayesian learner, the interacitve debugger module also requires a synthesizer to produce a series of trees for expert to label, so that the expert will finally identify a correct query answer. A key challenge here is that expert feedback is prohibitively expensive to obtain. Thus, the synthesizer must attempt to best leverage expert's feedback. We develop these methods in Section 5.3.2, one of which builds upon the previous greed method. We summarize our approaches in Section 5.3.3.

5.3.1 Bayesian Link Update

The framework of presenting a tree that the system considers to be correct, getting feedback from expert, and updating evidence of the tree being correct, naturally fits into the Bayesian approach. A Bayesian approach usually assumes some "prior" information and, when actual data is revealed, updates the underlying to obtain "posterior" information. Here we first formulate how to use Bayesian method to update probabilities associated with edges in a given tree, given their existing probabilities and the label of the tree. This approach can also be extended to handle *noisy oracle* and incorporate *confidence.*

5.3.1.1 Model Specification

We assume (1) edge independence and (2) that a tree receives positive expert feedback iff all of its edges are correct links. We further assume perfect feedback that is always correct (we consider the case of noisy feedback in Section 5.3.1.2). The challenge here is that one piece of feedback is applied to a mixed *collection* of good edges and bad edges. Hence, we must be able to update all edges in the collection altogether.

In this dynamic learning model, we denote by $\mathrm{Pr}_t(e)$ the probability of a link e being correct at time t. The global time stamp starts at 0 and each feedback will increment it by 1. When a tree T receives binary feedback, we will update $\mathrm{Pr}_t(e)$ to $\mathrm{Pr}_{t+1}(e)$ for each edge $e \in T$, using values $\{\mathrm{Pr}_t(e'), e' \in E(T)\}$. When T receives positive feedback, $\mathrm{Pr}_{t+1}(e)$ is updated to

$$\mathrm{Pr}[\text{e is correct} \mid \text{T is correct}] = 1,$$

since every edge must be correct in order to form a correct tree. On the other hand, when T receives negative feedback, $\mathrm{Pr}_{t+1}(e)$ is updated to

$$\mathrm{Pr}[\text{e is correct} \mid \text{T is incorrect}] = \frac{\mathrm{Pr}_t(e)(1 - \prod_{e_1 \in T, e_1 \neq e} \mathrm{Pr}_t(e_1))}{1 - \prod_{e_2 \in T} \mathrm{Pr}_t(e_2)}$$

As a sanity check, it is clear that $\mathrm{Pr}_{t+1}(e) \leq \mathrm{Pr}_t(e)$ when e belongs to a false tree. In practice, if $1 - \prod_{e_2 \in T} \mathrm{Pr}_t(e_2) = 0$, our **Q** System first relaxes $\mathrm{Pr}_t(e_2)$ to $1 - \delta$ where δ is a small constant then applies the Bayesian formula.

5.3.1.2 Incorporating Noisy Oracle

The above formulation assumes perfect expert feedback. We now extend this model to the case where expert has a probability of ϵ to make a mistake. Suppose that a tree T receives positive feedback, then

$$\begin{aligned} \Pr_{t+1}(e) &= (1-\epsilon)\,\mathrm{Pr}[\text{e is correct} \mid \text{T is correct}] \\ &+ \epsilon\,\mathrm{Pr}[\text{e is correct} \mid \text{T is incorrect}]. \end{aligned}$$

Similarly, when a tree T receives negative feedback, then

$$\begin{aligned} \Pr_{t+1}(e) &= (1-\epsilon)\,\mathrm{Pr}[\text{e is correct} \mid \text{T is incorrect}] \\ &+ \epsilon\,\mathrm{Pr}[\text{e is correct} \mid \text{T is correct}]. \end{aligned}$$

This model has a natural connection to the Steiner tree model used in keyword search over structured data and search-driven integration. Indeed, if each edge is associated with a cost equal to the negative loglikelihood of its probability, then the cost of a tree (which is the sum of all edge costs) will exactly be the negative loglikelihood of this tree being correct [77, 86].

5.3.1.3 Incorporating Confidence

Depending on query results and/or expertise of the data analyst, feedback may vary in confidence. This, ideally, should be reflected in the update formula as well. In fact, suppose that the data expert specifies a confidence value $0 < c \leq 1$. To achieve the confidence at the required level, for any given tree T, we repeatedly apply the above Bayesian formula until $\Pr[T] > c$ if feedback is positive or Pr[T is incorrect] $> c$ if feedback is negative.

5.3.1.4 Summary: Bayesian Update

Algorithm 8 Bayesian Update.

Input: Tree T with expert label L, expert confidence c, probability of expert error ϵ, current edge probabilities $\Pr_t$

Output: Updated edge probabilities $\Pr_{t+1}$

```
1: while $\Pr[T \text{ matches } L] < c$ do
2:     Apply Bayesian formula in Section 5.3.1.2
3: end while
4: return $\Pr_{t+1}$
```

Algorithm 8 puts the above models together and describes the Bayesian approach.

5.3.2 Synthesizing Labeling Samples

We have described above a formal framework for updating edge weights with respect to expert feedback. Unfortunately, it is not always easy to identify, for the purpose of query debugging, which tree(s) should be selected for expert to label, for two reasons: (1) expert feedback is usually expensive to obtain; (2) there are exponential numbers of candidate answers for a given keyword query. Therefore, a challenge for the debugger module is a strategy for *selecting* a series of trees to best leverage expert feedback interactively.

This problem setting is orthogonal to computing relevant answers in keyword-search-over-structured-data. In fact, we compute top-k Steiner trees for a given keyword query, which are mostly adopted in the literature [88]. We assume that top answers are incorrect and hence user enters the interactive debugging mode. Thus, for a given keyword query, we focus on strategies for selecting samples for experts to label. But these samples are not required to be Steiner trees connecting leaf nodes that match keywords, but rather informative partial answers that help the system get feedback and update underlying graph.

We consider two selection strategies. The first method (Section 5.3.2.1) directly computes alternative Steiner trees for the same query, but uses different measures other than predicted probability. The second method (Section 5.3.2.2) is mostly similar to selecting links mostly shared in the boolean model – it selects links that appear in a large number of query results. We describe both methods below.

5.3.2.1 Reducing Uncertainty

Our first selection strategy is motivated by uncertainty sampling in active learning, which seeks to select samples that can mostly alter the underlying model. This class of methods is shown to be highly effective in the literature when there is limit on the amount of feedback obtained.

Here, we use the notion of expected model change, defined as the expected amount of uncertainty reduction. At any point of time, we can measure the amount of uncertainty of an edge e using entropy. By assuming edge independence, we can further compute the uncertainty of a tree by summing up entropy values of individual edges. In addition, for an unlabeled tree T with probability p being correct, assuming membership oracle has error rate ϵ, with probability $(1-\epsilon)p+\epsilon(1-p)$ tree T receives positive feedback, with probability $\epsilon p + (1-\epsilon)(1-p)$ tree T receives negative feedback. Furthermore, we can simulate either positive feedback or negative feedback to compute tree entropy after feedback. The difference between before-feedback entropy and expected after-feedback entropy is, therefore, the expected model change. In fact, it is always non-negative when $\epsilon = 0$, meaning that the amount of model uncertainty keeps reducing.

Proposition 5 *When $\epsilon = 0$, the expected amount of uncertainty reduction is always non-*

negative.

Using this debugging strategy, within the top-k Steiner trees, a tree with the most amount of expected model change will be selected for receiving expert feedback, so that its edge weights will be updated. This feedback-loop continues until a correct tree is found.

5.3.2.2 Recursive Search based on Edge Frequency

Alternatively, our second selection is similar to labeling the mostly shared link in the boolean model 5.2.3. The selection criteria is the same. Edges that show up frequently in query answers will be prioritized for getting feedback on.

5.3.3 Putting it All Together

Algorithm 9 Debugging Weighted Graph.

Input: Set of query trees $A = \{T_i\}$, user U, keyword query KQ, input graph G.
Output: Updated graph.

```
1: while $U$ is not exhausted do
2:     Select a tree $T$ with the largest expected model change(Section 5.3.2.1) or an edge
       $T = \{e\}$ with the largest frequency(Section 5.3.2.2).
3:     Obtain label of $T$ and apply the Bayesian method to update edge weights in the
       graph.
4:     If recomputing answers for $KQ$ on $G$ yields satisfactory results, stop
5: end while
```

We summarize our approaches for debugging each keyword query.

1. Find the top-k Steiner trees and enter the debugging mode if none is satisfactory.

2. Repeatedly select samples for labeling, using either Section 5.3.2.1 or Section 5.3.2.2, until expert is exhausted or a correct tree is found.

This is illustrated in Algorithm 9.

5.4 Experimental Analysis

In this section, we describe in detail our experimental design and analysis. Our goal is to validate effectiveness of our proposed methods over diverse real world datasets requiring automatic record linking and schema matching, with different sizes in different domains (details are presented in Section 5.4.1). In addition, we evaluate the effectiveness and understand the differences among various debugging approaches.

Specifically, we conduct our experiments in the keyword-search-driven framework [77], where data is represented as a graph (with nodes linked *within* data sources via existing structural information, and with nodes linked *across* data sources via automated record linking algorithms; note that in both cases the links may be incorrect). After a dataset is loaded, a pool of keyword queries will be issued in a sequence, each generating some ranked query results. If the top query results are not satisfactory according to a gold-standard (as defined by the raw source data or a human panel), this triggers *debugging mode* and invokes the algorithms we study in this chapter. During debugging, we execute one of our proposed methods until some termination condition occurs, e.g., the top recomputed answers match the gold standard, or too much feedback has been requested (i.e., the user is exhausted). We describe detailed setup and methodology in Section 5.4.1.

Given the above evaluation flow, we wish to answer the following questions:

1. Are our methods *efficient* at debugging a given keyword query, in general? More precisely, can they quickly identify and remove false links and discover correct results?

2. Do graph repairs (altering link classification in the boolean model or changing edge probability in the weighted model) benefit other keyword queries as well, due to common edges? Can our proposed graph updates *distinguish* good links from bad ones, given enough amount of feedback?

3. Does search-driven integration benefit from the addition of the debugging module?

Observe that we assume a user can typically identify whether a result is a plausible answer to his or her query, as is often true in practice (particularly with data-expert or scientist users). This *query-driven* assessment of answers is easier for many users than a *query independent* question, e.g., about whether two data values should be linked, for any generic query.

With these questions in mind, we outline a set of metrics and report evaluation results. We present our analysis for the Boolean edge model in Section 5.4.2, and for the weighted-edge model in Section 5.4.3, respectively. There, we also disscuss the benefits of the debugging module over prior learning approaches in search-driven integration. We discuss the results and briefly comment on the difference between the two models in Section 5.4.4.

5.4.1 Overview: Datasets and Methodology

Platform. We implement our algorithms within our **Q** System [86], which contains roughly 60k lines of Java code. We conduct the experiments on an Intel Xeon CUP (2.83GHz, 2 processors) Windows Server 2008 (64-bit) machine with 24GB RAM, using 64-bit Java SE 1.60.11. We leverage **Q**'s capability to return a ranked list of answers for a given keyword search and its ability to interact with user feedback in real time.

Datasets. We wish to cover a variety of real datasets in different domains. One difficulty of choosing datasets for keyword search experiments is that, it is difficult (and sometimes impossible) to derive the set of correct links and therefore query answers. As one objective baseline, we first choose the popular **IMDB** database, for which links are known (encoded in the database as foreign keys). Our experiment hides information from the system about these correct links, and seeks to recover them automatically, from record linking algorithms and the debugging process. Our second dataset integrates subsets of **IMDB** database and **DBpedia**, where we establish a set of weighted, thresholded schema alignments from **IMDB** attributes to **DBpedia** attributes (such as properties related to people and movies). Finally, we evaluate our approaches on the **Probase** concept hierarchy, which is automatically extracted from the Web, and hence inherently contains many noisy and incorrect links. For example, edges such as "nonprofit $\rightarrow$ social enterprise", "video.google.com $\rightarrow$ engine", and "internet $\rightarrow$ utility cost" seem reasonable, but edges such as "characteristic$\rightarrow$ inventory cost factor" do not seem very informative. **Probase** is the largest dataset in our evaluation (in terms of number of links). Although it is hard to derive a gold standard for correct links in **Probase**, we try to remain conservative when we claim a link to be false. We outline properties of these datasets below. The "number of possible edges" describes how many links our schema matchers claim to be true initially, which we will explain in more detail in the next paragraph. For **Probase**, since it is infeasible to examine all 14k links, the exact number of true alignments is unknown.

Name	**IMDB**	**IMDB + DBpedia**	**Probase**
No. Nodes	21	255	8,087,141
No. Possible edges	20	668	13,949,065
No. True alignments	9	14	N/A
Size (in MB)	1082	1417.8	1139

Methodology. For each dataset, the first stage is to predict a set of possible links. For **Probase**, this is simply the set of input links. For **IMDB** and **IMDB+DBpedia**, we run a set of schema matching primitives from the COMA++ library and obtain a value in $[0, 1]$ for every possible alignment. We prune links for which all matching primitives provide very low probabilities. The remaining edges are therefore initially classified as true. In addition, edge probabilities (averaged over all matchers) will be used as priors in the weighted model.

Query workload and feedback. Our query workload consists of 10 queries for each dataset, whose results are revisited (and new feedback is given) three times. The queries were based on common-knowledge searches. Sample queries include "academic, technology", "Shanghai, China", "Aaria, Final Fantasy".

5.4.2 Boolean Model

For each dataset, in the boolean model, we start from a graph where links may be falsely predicted. As described previously, these predictions (for **IMDB** and **IMDB+Dbpedia**) are obtained from applying COMA++ matchers followed by pruning links with very low probabilities. In fact, these optimistic predictions only contain one-sided error: a true link always receives positive prior classification. Then we simulate to issue each keyword query and enter the interactive debugging mode if the top-1 result obtained is not "correct". For **IMDB** and **IMDB+Dbpedia**, based on the provided gold standard, we mark a query result to be true if and only if all links in the tree are known to be correct. For **Probase**, we use real-world knowledge to judge result correctness, as there is no known gold standard. The system will invoke one of the debugging algorithms to select some edges for labeling, and update answers for the same keyword query until the top-1 result is satisfactory.

We compare three methods in our experiments.

1. (Baseline) Ask labeling of each edge in a false tree. These edge labels will be reused later whenever possible: if a candidate tree contains a false edge claimed earlier, it will not show up. Each edge will be requested for its label at most once.

2. Recursive search until a false link is confirmed, based on Section 5.2.2. This is triggered when a false tree is presented, and when a false edge is found, the **Q** System recomputes the top-1 tree.

3. Sort links based on how many times they appear in the top-k answers and request label for the most heavily shared link. This is described in Section 5.2.3. We term this method "frequency-based prioritization" and use $k = 10$.

Name	**IMDB**	**IMDB + DBpedia**	**Probase**
Baseline	0.6	3.3	8
Recursive Search	**0.2**	1.1	3.7
Frequency-based Prioritization	**0.2**	**0.6**	**2**

Table 7: Average number of labeling questions required to get a correct answer.

We conduct a set of experiments to measure how these three approaches perform on the three datasets. To start, our first experiment measures the number of labeling questions required to obtain a correct answer. This measures the "efficiency" of each of our methods – smaller numbers mean less feedback requested. During the process, feedback on recomputed top-1 results is not counted as labeling questions. We only count labeling questions in the debugging procedure.

The results for this experiment are shown in Table 7. As we see, both of our methods outperform the baseline and frequency-based method is slightly better. Note that for **IMDB**, all methods require very little feedback, since running schema matching algorithms cleans up a large portion of the input graph. But on **Probase** where noisy links are common, our proposed methods save on average 4-6 labeling questions.

Next, we proceed to measure if our methods can improve the system's ability to repair underlying graph. Towards this goal, in the third and fourth experiment, using the same 10 keyword queries, we apply the same number of labeling questions allowed (5 questions) for debugging each query. We measure, using the same total amount of feedback, number

Name	**IMDB**	**IMDB + DBpedia**	**Probase**
Baseline	(9, 2)	(5, 10)	(6,17)
Recursive Search	(9, 2)	(5, 16)	(8,19)
Frequency-based Prioritization	(9, 2)	(5, 20)	(11,24)

Table 8: Total number of true and false links confirmed, respectively, given the same amount of feedback on each dataset.

of edges confirmed to be true and false, respectively. We exclude *keyword matching* edges in the measurement.

We show the results for this set of experiment in Table 8. As illustrated, our proposed methods are able to remove more false edges than the baseline does. On **Probase**, our methods can also confirm more correct links. This verifies their abilities to repair the graph. Between our proposed methods, frequency-based method is slighter better.

5.4.3 Weighted Model

We now look at the weighted model. Like the boolean model, we wish to understand how our debugging methods can get correct answers and improve the graph in general. Here the evaluation process is similar to what we have used in the boolean model. However, here we must incorporate probability values, instead of directly using binary classification. Hence, for each keyword query, we rank the candidate answers (i.e. Steiner trees) based on their probabilities of being true.

Similar to the boolean model, we first use COMA++ matchers to predict link probabilities in **IMDB** and **IMDB+Dbpedia**. Unlike the boolean model, we will not threshold these probabilities and convert them to zero or one. Instead, we will use these probabilitie as they are to compute ranked answers. Simulation of feedback loop is the same. For each keyword query, the **Q** System invokes the interactive debugging model if the top-1 result obtained is not "correct". During debugging, the system applies one of the debugging algorithms to select a series of samples for labeing, resulting in updates on edge probabilities, until a recomputed top-1 result is satisfactory.

We compare three methods in our experiments.

1. (Baseline) Ask labeling of each edge in a false tree. This will assign a close-to-zero probability or a close-to-one probability, depending on feedback. These probabilities

will be reused. Each edge will be requested for its label at most once.

2. Apply Bayesian link update (Section 5.3.1) and select a sample with the largest expected model change (Section 5.3.2.1). We term this method **BayesianEMC**.

3. Apply Bayesian link update and select a link with the largest frequency (Section 5.3.2.2). We term this method **BayesianGreedy**.

Notice that although the Bayesian method allows us to specify probability of oracle making an error and oracle confidence, here we do not use these two parameters due to the challenge of defining a single gold standard.

Name	**IMDB**	**IMDB + DBpedia**	**Probase**
Baseline	0.6	3.3	8
BayesianEMC	0.4	1.7	3.6
BayesianGreedy	0.2	0.6	1.9

Table 9: Average number of labeling questions required to get a correct answer.

We now look at the efficiency of our debugging approaches and examine how many labeling questions are needed to get a correct answer. We show the results in Table 9. This shows that our proposed methods very effectively find correct answers and that they perform better than the baseline. We discover that **BayesianGreedy** is slightly better, since it is mostly efficient in lowering probabilities of those false trees containing heavily-shared false edges.

Our experiments have also shown that in noisy graphs such as **Probase**, debugging is able to aid user to find satisfactory results. This is hard to achieve in search-driven integration, where the system fails to learn from negative answers to a keyword query.

5.4.4 Discussion

Our experiments have demonstrated effectiveness of our proposed methods. Overall they provide noticeable improvements over naive debugging approaches. The advantage of the boolean model is precise classification and its debugging procedure is to identify one false edge either as fast as possible or as efficient in pruning search space as possible. On the other hand, the weighted model does not require strict classification and is perhaps a better model for modeling knowledge bases. Its debugging procedures aim to reduce model uncertainty as much as possible, but differ from the boolean model since the Bayesian method adjusts weights of multiple edges at once instead of pinpointing to a single problematic edge.

5.5 Conclusion

Motivated by debugging keyword search results in search driven data integration, we study the generalized problem of repairing links in automatically constructed knowledge bases and semi-supervised graphs, where graph links are initially predicted and then gradually fixed over time, by learning from expert feedback. We have studied in detail two learning models, the exact learning model and the approximate learning model. In these two models, we develop methods to select labeling samples and to update graph links based on expert feedback. Experiment results have demonstrated that our proposed methods can efficiently take advantage of user feedback to repair graph links. Building upon this work, we seek to study more generalized learning models, for instance, one that relaxes the edge independence assumption.

Chapter 6

Related Work

This thesis decomposes tasks in search-based integration into user-driven components and adapts learning techniques to solve them. This work is at thematically related to several threads of research in the database and machine learning communities. We discuss these related areas in this chapter.

6.1 Related Work in Data Management

6.1.1 Data Integration

We have already briefly commented prior work on conventional integration techniques mostly in Chapter 1. We provide more references here.

To actually obtain scores, **Q** system incorporates a suite of off-the-shelf schema matchers. Like most modern matchers [24, 28], the **Q** system combines output from multiple sub-matchers [24] (in particular we use their base matchers and learn to compose them). Our approach differs from many others [24], such as LSD [25] and COMA++ [24], by focusing on *online learning* given candidate answers for queries posed by the user. It is also notable that our approach eliminates the notion of a mediated schema, going directly to a user-driven query — which makes it quite different from the example-driven approach to learning schema mappings of ten Cate et al. [1, 17]. The focus of thesis is on a general *architecture* for incorporating the output of matchers while obtaining entropy information. A key contribution of the work in Chapter 3 is to create, then compose across a query tree, a probability distribution based on (weighted) outputs of the different matchers. The problem of modeling uncertainty in schema matching is discussed in [58].

A variety of vision papers have proposed pay-as-you-go or dataspaces models for integration [22, 30], and some aspects of that vision were implemented in iTrails [81]. However, many aspects of achieving the pay-as-you-go vision have remained open. We argue that the **Q** System provides some building blocks towards the overall goals.

More generally, data integration and its theoretical counterpart data exchange are ongoing research topics in the database community. Most recent surveys can be found in [5] and [26].

6.1.2 Keyword Search over Structured Data and Top-k Query Answering

We have briefly outlined related work on keyword search over structured data in Chapter 1 and Chapter 2. The **Q** System extends the basic approach of BANKS, BLINKS, DISCOVER, and XRANK [13, 39, 43, 50] originally proposed for centralized databases, to a model where the data is distributed across many sources and where answers' scores must be learned [48, 76, 77]. Later systems also explored keyword search as a model for data integration [11, 66]. The key difference in our work on the **Q** System is the iterative loop through *end-user feedback on query answers.*

To return effective answers at scale, our query answering subsystem [48, 77] must efficiently compute approximate top-k query answers according to the scoring model. The general top-k problem has been studied in a variety of contexts [29, 46, 54, 57], and our system leverages many ideas from that literature. However, as noted in Chapter 3, the key challenge for active learning lies in the fact that parts of our scoring function are not decomposable.

6.1.3 Learning Queries and Mappings

Belhajjame et al developed a model to group users into *feedback clusters* [9] that gets similar ranking functions; this relies on overlap among the query load, unlike our more general collaborative learning that relies on shared *features.* Most current search-driven integration systems make use of *online learning* given candidate *answers to queries* posed by the user – as opposed to query-independent schema matching techniques, most notably COMA++ [24] and recent work on pay-as-you-go schema matching by communities [62]. Recent work has studied example-driven approaches to learning mappings, such as that of ten Cate et al. [79], which attempts to learn *schema mappings* in a mediated schema. Recent work has also explored learning queries from data [71], but our approach is focused on learning to rank output trees as opposed to query expressions.

6.1.4 Crowdsourcing

We have already pointed out how this thesis work is related to and different from recent work in crowdsourcing in Chapter 1. Here we provide some additional comments.

Learning in search-driven integration can be viewed as a form of "expert"-sourcing or crowdsourcing. Crowdsourced techniques for finding record links, using Mechanical Turkers and others, have been shown to be quite effective [82], and in fact most of the strategies used here are closely related to active learning. Entity resolution or record linking in these settings is typically done in a query-independent fashion. A variety of schemes relied on voting to resolve conflicting input, though several machine learning approaches to the problem [52, 56, 91] use probabilistic techniques such as expectation maximization or belief propagation to estimate user expertise and reliability. In contrast our approach to collaborative learning does not assume a single "true" value.

Crowdsourced routing has been recently studied [72, 90], in which the notion of uncertainty reduction has also been applied. However, the focus there is on finding the *best* path between a *pair* of source and destination nodes, which is different from ours. An additional key difference lies in our development of learning from expert feedback. On the other hand, the question studied in crowdsourced graph search [63] is also very different from ours. In that work, the goal is to recover all "target nodes" from a given source by asking reachability questions.

6.1.5 Others

The problem of selecting the *most informative* feedback has been studied in data cleaning [84] and data integration. For integration, approaches include focusing on highest-value candidate schema matches for dataspaces [49] and on active learning for refining record linking [4]. These are related in spirit to our approach, but they get feedback over *individual alignments* whereas our active learning technique seek to understand the uncertainty associated with an entire query, and combine high-scoring and uncertain queries' results.

Methods for incorporating diversity into top-k query answers have been studied extensively in the information retrieval context. Gollapudi et al [35] developed several measures in addition to the max-sum diversification scheme we adopt, which emphasize factors such as minimum amount of diversity instead of combined diversity. An excellent survey of appears in [27]. Deng and Fan [23] also study the complexity of result diversification.

6.1.6 Debugging keyword query

Closely related to the query debugging chapter is recent work on debugging incorrect results in keyword search systems [7]. That paper proposes efficient search strategies for finding a maximal query tree that returns non-empty answers with respect to a given "false" query tree. However, their model assumes known relations and links in a static schema graph. Our work goes beyond that by incorporating a dynamic cost model in a wider range of applications.

6.1.7 Why and why-not

Why and how a tuple contributes to query answers have been extensively studied in context of provenance [36], responsibility [59], and formal explanations [65]. By contrast, there has been increasing amount of work on modeling and computing explanation of non-answers in the database community. These approaches can be roughly classified into three categories. **Data-centric** methods propose *modifications* to the input database, often based on the notion of provenance, so that running the same query over the updated database yields the missing answers [41, 44, 59]. Alternatively, **query-centric** approaches attempt to modify the input query so that the new query produces the missing answer [18, 80].

Finally, **ontology-based** approaches rely on auxilary external ontologies to compute explanations [34, 78]. Such ontologies model relationships among schema elements and are usually provided externally or inferred from schema. There are also studies on why-not explanations in the context of top-k queries [19, 33, 40]. The goals of our work are different and consist of determining where data experts should focus on and helping data experts to debug incorrect answers, by learning a dynamic cost model. Recent work Data X-Ray [83] with a similar cost model and Bayesian update method is also close related. The different lies in learning settings and generalization of the the Bayesian method.

6.2 Related Work in Machine Learning

6.2.1 Active Learning

Active learning attempts to address the issue of high labeling cost. Typically, in active learning, a learning algorithm has access to unlabeled data and has the ability to select the next (explicit) sample for an oracle's annotation. Requesting the next labeled sample can either be done by explicitly constructing the sample or by issuing a query for a highly informative sample, depending on different learning models. The objective of active learning is to learn a good model with significantly smaller number of samples requested.

While active learning is a popular area of machine learning [68], standard techniques cannot be directly used on tree-structured queries in which individual edges have uncertainty. Three strategies have been primarily used in prior research: (1) the least confident strategy considers only the most likely prediction; (2) the maximum margin strategy considers the top two predictions; (3) the entropy maximization strategy considers all predictions, which can be exponential in the size of the structured object predicted. Our approach of clustering predictions and choosing a representative tree per cluster is, in some sense, an intermediate strategy.

Prior work on active learning over structured output has sought to select the next *instance* upon which to receive feedback, with feedback directly over the predicted objects [68, Sec. 2.4]. Our work differs in keeping the instance (keyword query) the same, and soliciting feedback over different trees (interpretations) of the given query. With the notable exception of [45], most previous work on active learning over structured output involved sequences [21, 67], whereas in the **Q** system we infer trees. Also, our use of active learning in the keyword search-based data integration is novel.

Although cluster-based active learning has been found to be useful in previous research [68, Sec. 5.2], such work has focused on classification and not structured prediction. Moreover, clustering in those cases is performed over the input instances, rather than the output Steiner trees corresponding to the given keyword query.

Another thread of related work is active learning methods on trees and graphs [**?**], which investigates optimal arrangement of queries to minimize mistakes on non-queries nodes. It uses spanning tree-based query selection methods and provides bounds on number of mistakes. This work assumes a different model from ours, in which it predicts *binary labels on nodes*.

6.2.2 Recommendation Systems

The rise of recommendation systems, especially of their techniques, has drawn much attention from both academia and industry: there have been tons of related techniques developed in this area and many modern sites like `Amazon` and `Netflix` deploy recommendation systems to pursue more profit. The goal of a recommendation system is to predict how likely a user will prefer an item, based on past history.

Recommendation systems [51] are typically classified into three main categories: content-based methods, collaborative filtering methods, and hybrid methods. Content-based methods attempt to recommend items to a user based on what kind of items the user likes the most, by computing item similarities and examining user's past history. Alternatively, content-based methods may also model user profiles to discover similar users. Content-based recommendation methods are often limited since they only consider the history of the particular user, largely ignoring correlations within the huge set of users. To this extent, recent methods mostly build on collaborative models, which aim to leverage ratings from other users as well. Collaborative filtering methods are further divided into neighborhood based-approaches and model-based approaches. A neighborhood based- (or memory based-) method usually predicts a rating based on an aggregated score derived from the entire or a subset of the ratings (for example, ratings from similar users). In contrast to neighborhood-based methods, model-based methods seek to build effective statistical models that best reflect the rating behaviors and patterns. These models are trained from existing data and will be applied to predict unknown ratings. Examples of such models include Bayesian networks, clustering, regressions, and Singular Value Decomposition (SVD). Recent work [10] shows that neighborhood-based methods are strong at discovering local structure but weak at predicting overall ratings while the opposite holds for model-based approaches. Furthermore, in a hybrid approach, one seeks to utilize content data and to build a unified model incorporating both content-based and collaborative methods.

Our work in Chapter 4 builds upon, that of collaborative filtering [8, 51, 73], where the goal is to develop personalized rankings of items, based on their similarities to other users, and those users' preferences. Our work has a more difficult problem than traditional collaborative filtering, in that the basic items we seek to rank — query trees — have structure and may overlap with one another. We also have close ties between the collaborative filtering and online learning aspects of our platform. These have been the focal points of study in Chapter 4.

6.2.3 Learning by Membership Query Synthesis

Our work on query debugging, and more generally the vision of improving integration quality by *synthesizing* examples for labeling, is very similar to learning by *membership query synthesis*. In this setting, the learner *creates* the sample on demand itself and requests sample label from an oracle. This model has been heavily studied in the literature [2, 3, 53]. To the best our knowledge, our proposed learning models differ from those and are more complex.

Chapter 7

Conclusion and Future Work

7.1 Summary

The vision of rapid information integration remains elusive. Recent work has proposed to complement (or even replace) conventional integration with a "pay-as-you-go", keyword search-driven data integration model. This thesis addresses several fundamental research challenges in implementing this model in an end-to-end system, where integration is driven by users' information needs specified as keywords, and integration quality is iteratively improved from user feedback given onto query results. These challenges require novel solutions to combine learning and limited amount of expert feedback to best improve integration, sometimes in very noisy models. Overall, this thesis proposes

- Active learning techniques to repair links from small amounts of user feedback;
- Collaborative learning techniques to combine users' conflicting feedback;
- Debugging techniques to identify where data experts could best improve integration quality.

In developing these methods, this thesis also describes several basic building blocks applicable to global-scale data integration:

- Combing outputs from schema matching and/or record linking tools to estimate the amount of uncertainty associated with a query results;

- Active learning-based methods to sample relevant results with high uncertainty to best improve the system's ability to learn, such as uncertainty sampling and ranking by expected model change.

- Means of diversifying query results, through clustering or optimizing a diversity objective function, so that user feedback can be more informative.

- Directly incorporating user similarity into online learning algorithm, so that feedback from one user can propagate to other users as well.

- Matrix factorization-based collaborative filtering methods that decompose users and integration features into smaller spaces, to predict the relevance of any pair of user and feature.

- Debugging queries through sample synthesis to improve integration over large noisy data, and two formal models for this approach: exact boolean model and approximate weighted model.

- Recursive search method to quickly identify false links and greedy method to prune the search space of query answers, applied in the exact learning model.

- Bayesian method to update the system's belief on link probabilities and prioritization strategies to select the best partial answers to ask for user feedback, applied in the weighted model.

These methods are implemented within the **Q** System, a prototype of search-driven integration. Their effectiveness are validated over several real-world datasets, in a variety of domains such as bio-informatics and movies, through both synthetic experiments and real user studies.

7.2 Directions for Future Work

Within the huge problem space of search-driven integration, this thesis has only explored a starting set. Building upon this thesis, there are several possible directions for future work. We point out some of them below.

7.2.1 More Expressive Queries

The **Q** System returns Steiner trees as query answers to keyword search over structured data. This is widely adopted in the literature. But there are other alternatives [88] to Steiner trees as well, such as cliques under certain constraints. Similar to Steiner trees, some alternative semantics also generate unions of conjunctive queries. A natural question is how to learn from feedback on these types of query results.

Unions of conjunctive queries address a large class of integration tasks. Further along this direction, we also wish to examine a broader class of queries, including negation and aggregation. There is a question of how to enable users to specify such queries as well as how to learn from these query results. This direction might be much more challenging.

7.2.2 Mapping Expressiveness

So far, the **Q** System has focused on correspondence between data items and matchings between table attributes. These are much more express mappings that can be specified between two datasets, heavily studied in the literature of schema mapping and data exchange. There are a series of open research problems of learning such mappings in the search-driven integration framework. For example, what mappings can be learned? What can be efficiently learned? How to learn from user feedback?

7.2.3 Relaxing Feature Independence Assumption

To this end we have assumed that features are independent. This is an effective assumption but may not always be true. It might be beneficial to adopt a layered model in which features are clustered.

7.2.4 Theoretical Analysis

This thesis have provided extensive experimental analysis, sometimes with provable guarantee. More theoretical analysis will be desirable for a better understanding of search-driven integration. Several directions in this space include, for example, Probably-Approximately-Correct(PAC)-style analysis [53] and error bounds based on the notion of“no-regret”. In addition, one can study the case of "adversarial matchers" that produce bad matching scores. A major challenge here is the combinatorial nature of the problem.

7.2.5 Enabling Hypothetical Analysis

Currently, the **Q** System does not support "unlearning" from a feedback, i.e., reversing changes made to the model due to learning from feedback. While this seems achievable, more generally, we wish to allow user perform *hypothetical analysis of features* in the **Q** System. For instance, users may mark a set of features as "bad" all at once and wish to see how top-k results in a view change. As future work, we seek to leverage recent work on provenance of linear algebra [85] and enable interactive analysis of this kind.

7.2.6 More Evaluation

We have tested some parts of the system within the `www.ieeg.org` neuroscience portal [47]. Ultimately we wish to also study performance on a broad class of "knowledge graphs" like Freebase, YAGO [75], DBpedia [6], DBLP, and Nell [16]. We also seek to integrate these datasets and others on Linked Open Data.

Bibliography

[1] Bogdan Alexe, Balder Ten Cate, Phokion G Kolaitis, and Wang-Chiew Tan. Designing and refining schema mappings via data examples. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 133–144. ACM, 2011.

[2] Dana Angluin. Queries and concept learning. *Journal of Machine learning*, 2(4):319–342, 1988.

[3] Dana Angluin. Queries revisited. *Theoretical Computer Science*, 313(2):175–194, 2004.

[4] Arvind Arasu, Michaela Götz, and Raghav Kaushik. On active learning of record matching packages. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 783–794, 2010.

[5] Marcelo Arenas, Pablo Barceló, Leonid Libkin, and Filip Murlak. *Foundations of Data Exchange*. Cambridge University Press, 2014.

[6] Soren Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. Dbpedia: A nucleus for a web of open data. In Karl Aberer, Key-Sun Choi, Natasha Noy, Dean Allemang, Kyung-Il Lee, Lyndon Nixon, Jennifer Golbeck, Peter Mika, Diana Maynard, Riichiro Mizoguchi, Guus Schreiber, and Philippe Cudr?Mauroux, editors, *The Semantic Web*, volume 4825 of *Lecture Notes in Computer Science*, pages 722–735. Springer Berlin Heidelberg, 2007.

[7] Akanksha Baid, Wentao Wu, Chong Sun, A Doan, and Jeffrey F Naughton. On debugging non-answers in keyword search systems. In *International Conference on Extending Database Technology (EDBT)*, 2015.

[8] Nicola Barbieri, Giuseppe Manco, and Ettore Ritacco. Probabilistic approaches to recommendations. *Synthesis Lectures on Data Mining and Knowledge Discovery*, 5(2):1–197, 2014.

[9] Khalid Belhajjame, Norman W Paton, Cornelia Hedeler, and Alvaro AA Fernandes. Enabling community-driven information integration through clustering. *Distributed and Parallel Databases*, 33(1):33–67, 2015.

[10] Robert M Bell and Yehuda Koren. Lessons from the netflix prize challenge. *ACM SIGKDD Explorations Newsletter*, 9(2):75–79, 2007.

[11] Sonia Bergamaschi, Elton Domnori, Francesco Guerra, Raquel Trillo Lado, and Yannis Velegrakis. Keyword search over relational databases: a metadata approach. In *Proceedings of the ACM SIGMOD International Conference on Management of data*, pages 565–576, 2011.

[12] Philip A. Bernstein and Laura M. Haas. Information integration in the enterprise. *Commun. ACM*, 51(9):72–79, 2008.

[13] Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, and Shashank Sudarshan. Keyword searching and browsing in databases using banks. In *IEEE International Conference on Data Engineering (ICDE)*, pages 431–440. IEEE, 2002.

[14] Angela Bonifati, Radu Ciucanu, Aurélien Lemay, and Sławek Staworko. A paradigm for learning queries on big data. In *Proceedings of the First International Workshop on Bringing the Value of Big Data to Users (Data4U 2014)*, page 7. ACM, 2014.

[15] Sarah Cohen Boulakia, Olivier Biton, Susan B. Davidson, and Christine Froidevaux. BioGuideSRS: querying multiple sources with a user-centric perspective. *Bioinformatics*, 2007.

[16] Andrew Carlson, Justin Betteridge, Bryan Kisiel, Burr Settles, Estevam R Hruschka Jr, and Tom M Mitchell. Toward an architecture for never-ending language learning. In *AAAI*, volume 5, page 3, 2010.

[17] Balder Ten Cate, Víctor Dalmau, and Phokion G Kolaitis. Learning schema mappings. *ACM Transactions on Database Systems (TODS)*, 38(4):28, 2013.

[18] Adriane Chapman and HV Jagadish. Why not? In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 523–534, 2009.

[19] Lei Chen, Xin Lin, Haibo Hu, Christian S Jensen, and Jianliang Xu. Answering why-not questions on spatial keyword top-k queries. In *IEEE International Conference on Data Engineering (ICDE)*, pages 279–290, 2015.

[20] Koby Crammer, Ofer Dekel, Joseph Keshet, Shai Shalev-Shwartz, and Yoram Singer. Online passive-aggressive algorithms. *The Journal of Machine Learning Research*, 7:551–585, 2006.

[21] Aron Culotta and Andrew McCallum. Reducing labeling effort for structured prediction tasks. In *AAAI*, pages 746–751, 2005.

[22] Anish Das Sarma, Xin Dong, and Alon Halevy. Bootstrapping pay-as-you-go data integration systems. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 861–874. ACM, 2008.

[23] Ting Deng and Wenfei Fan. On the complexity of query result diversification. *Proc. VLDB Endow.*, 6(8), June 2013.

[24] Hong-Hai Do and Erhard Rahm. Matching large schemas: Approaches and evaluation. *Information Systems*, 32(6):857–885, 2007.

[25] AnHai Doan, Pedro Domingos, and Alon Y Halevy. Reconciling schemas of disparate data sources: A machine-learning approach. *ACM Sigmod Record*, 30(2):509–520, 2001.

[26] AnHai Doan, Alon Halevy, and Zachary Ives. *Principles of data integration*. Elsevier, 2012.

[27] Marina Drosou and Evaggelia Pitoura. Search result diversification. *SIGMOD Record*, 39(1), September 2010.

[28] Ahmed K Elmagarmid, Panagiotis G Ipeirotis, and Vassilios S Verykios. Duplicate record detection: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 19(1):1–16, 2007.

[29] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences*, 66(4):614–656, 2003.

[30] Michael Franklin, Alon Halevy, and David Maier. From databases to dataspaces: a new abstraction for information management. *ACM Sigmod Record*, 34(4):27–33, 2005.

[31] Michael J Franklin, Donald Kossmann, Tim Kraska, Sukriti Ramesh, and Reynold Xin. Crowddb: answering queries with crowdsourcing. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 61–72, 2011.

[32] Avigdor Gal and Tomer Sagi. Tuning the ensemble selection process of schema matchers. *Information Systems*, 35(8):845–859, 2010.

[33] Yunjun Gao, Qing Liu, Gang Chen, Baihua Zheng, and Linlin Zhou. Answering why-not questions on reverse top-k queries. *Proceedings of the VLDB Endowment*, 8(7):738–749, 2015.

[34] Boris Glavic, Sven Köhler, Sean Riddle, and Bertram Ludäscher. Towards constraint-based explanations for answers and non-answers. In *Proceedings of the USENIX Conference on Theory and Practice of Provenance*, pages 13–13, 2015.

[35] Sreenivas Gollapudi and Aneesh Sharma. An axiomatic approach for result diversification. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, 2009.

[36] Todd J Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 31–40, 2007.

[37] Fan Guo, Chao Liu, Anitha Kannan, Tom Minka, Michael J. Taylor, Yi Min Wang, and Christos Faloutsos. Click chain model in web search. In *WWW*, pages 11–20, 2009.

[38] Alon Y Halevy, Zachary G Ives, Dan Suciu, and Igor Tatarinov. Schema mediation in peer data management systems. In *IEEE International Conference on Data Engineering (ICDE)*, pages 505–516, 2003.

[39] Hao He, Haixun Wang, Jun Yang, and Philip S Yu. Blinks: ranked keyword searches on graphs. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 305–316, 2007.

[40] Zhian He and Eric Lo. Answering why-not questions on top-k queries. *TKDE*, 26(6):1300–1315, 2014.

[41] Melanie Herschel, Mauricio A Hernández, and Wang-Chiew Tan. Artemis: A system for analyzing missing answers. *Proceedings of the VLDB Endowment*, 2(2):1550–1553, 2009.

[42] Thomas Hofmann. Latent semantic models for collaborative filtering. *ACM Transactions on Information Systems (TOIS)*, 22(1), 2004.

[43] Vagelis Hristidis and Yannis Papakonstantinou. Discover: Keyword search in relational databases. In *Proceedings of the international conference on Very Large Data Bases*, pages 670–681, 2002.

[44] Jiansheng Huang, Ting Chen, AnHai Doan, and Jeffrey F Naughton. On the provenance of non-answers to queries over extracted data. *Proceedings of the VLDB Endowment*, 1(1):736–747, 2008.

[45] Rebecca Hwa. Sample selection for statistical parsing. *Computational Linguistics*, 30(3):253–276, 2004.

[46] Ihab F Ilyas, Walid G Aref, and Ahmed K Elmagarmid. Supporting top-k join queries in relational databases. *The VLDB Journal*, 13(3):207–221, 2004.

[47] Zachary G Ives, Zhepeng Yan, Nan Zheng, Brian Litt, and Joost B Wagenaar. Looking at everything in context. *The 7th biennial Conference on Innovative Data Systems Research (CIDR)*, 2015.

[48] Marie Jacob and Zachary Ives. Sharing work in keyword search over databases. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 577–588, 2011.

[49] Shawn R Jeffery, Michael J Franklin, and Alon Y Halevy. Pay-as-you-go user feedback for dataspace systems. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 847–860. ACM, 2008.

[50] Varun Kacholia, Shashank Pandit, Soumen Chakrabarti, S. Sudarshan, Rushi Desai, and Hrishikesh Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, pages 505–516, 2005.

[51] Paul B Kantor, Lior Rokach, Francesco Ricci, and Bracha Shapira. *Recommender systems handbook*. Springer, 2011.

[52] David R. Karger, Sewoong Oh, and Devavrat Shah. Iterative learning for reliable crowdsourcing systems. In J. Shawe-Taylor, R.S. Zemel, P.L. Bartlett, F. Pereira, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 1953–1961. Curran Associates, Inc., 2011.

[53] Michael J Kearns and Umesh Virkumar Vazirani. *An introduction to computational learning theory*. MIT press, 1994.

[54] Benny Kimelfeld and Yehoshua Sagiv. Finding and approximating top-k answers in keyword proximity search. In *Proceedings of the ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 173–182, 2006.

[55] Brian Litt, Greg Worrell, and Zachary G. Ives. The international epilepsy electrophysiology portal. `www.ieeg.org`.

[56] Qiang Liu, Jian Peng, and Alex Ihler. Variational inference for crowdsourcing. In *Advances in Neural Information Processing Systems*, pages 692–700, 2012.

[57] Amélie Marian, Nicolas Bruno, and Luis Gravano. Evaluating top-k queries over web-accessible databases. *ACM Transactions on Database Systems (TODS)*, 29(2):319–362, 2004.

[58] Anan Marie and Avigdor Gal. Managing uncertainty in schema matcher ensembles. In *SUM*, pages 60–73, 2007.

[59] Alexandra Meliou, Wolfgang Gatterbauer, Katherine F Moore, and Dan Suciu. The complexity of causality and responsibility for query answers and non-answers. *Proceedings of the VLDB Endowment*, 4(1):34–45, 2010.

[60] Andriy Mnih and Ruslan Salakhutdinov. Probabilistic matrix factorization. In *Advances in Neural Information Processing Systems*, pages 1257–1264, 2007.

[61] Chad L Myers and Olga G Troyanskaya. Context-sensitive data integration and prediction of biological networks. *Bioinformatics*, 23(17):2322–2330, 2007.

[62] Quoc Viet Hung Nguyen, Thanh Tam Nguyen, Zoltán Miklós, Karl Aberer, Avigdor Gal, and Matthias Weidlich. Pay-as-you-go reconciliation in schema matching networks. In *IEEE International Conference on Data Engineering (ICDE)*, pages 220–231, 2014.

[63] Aditya Parameswaran, Anish Das Sarma, Hector Garcia-Molina, Neoklis Polyzotis, and Jennifer Widom. Human-assisted graph search: it's okay to ask questions. *Proceedings of the VLDB Endowment*, 4(5):267–278, 2011.

[64] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming.* Elsevier, 2006.

[65] Sudeepa Roy and Dan Suciu. A formal approach to finding explanations for database queries. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of data*, pages 1579–1590, 2014.

[66] Mayssam Sayyadian, Hieu LeKhac, AnHai Doan, and Luis Gravano. Efficient keyword search across heterogeneous relational databases. In *IEEE International Conference on Data Engineering (ICDE)*, pages 346–355, 2007.

[67] B. Settles and M. Craven. An analysis of active learning strategies for sequence labeling tasks. In *EMNLP*, 2008.

[68] Burr Settles. *Active Learning.* Morgan & Claypool, 2012.

[69] Burr Settles, Mark Craven, and Soumya Ray. Multiple-instance active learning. In *NIPS*, 2007.

[70] Si Shen, Botao Hu, Weizhu Chen, and Qiang Yang. Personalized click model through collaborative filtering. In *WSDM*, pages 323–332, 2012.

[71] Sławek Staworko and Piotr Wieczorek. Learning twig and path queries. In *Proceedings of the 15th International Conference on Database Theory*, pages 140–154. ACM, 2012.

[72] Han Su, Kai Zheng, Jiamin Huang, Hoyoung Jeung, Lei Chen, and Xiaofang Zhou. Crowdplanner: A crowd-based route recommendation system. In *IEEE International Conference on Data Engineering (ICDE)*, pages 1144–1155, 2014.

[73] Xiaoyuan Su and Taghi M Khoshgoftaar. A survey of collaborative filtering techniques. *Advances in artificial intelligence*, 2009.

[74] Fabian Suchanek, James Fan, Raphael Hoffmann, Sebastian Riedel, and Partha Pratim Talukdar. Advances in automated knowledge base construction. *SIGMOD Records, March*, 2013.

[75] Fabian M Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: A large ontology from wikipedia and wordnet. *Web Semantics: Science, Services and Agents on the World Wide Web*, 6(3):203–217, 2008.

[76] Partha Pratim Talukdar, Zachary G Ives, and Fernando Pereira. Automatically incorporating new sources in keyword search-based data integration. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 387–398, 2010.

[77] Partha Pratim Talukdar, Marie Jacob, Muhammad Salman Mehmood, Koby Crammer, Zachary G Ives, Fernando Pereira, and Sudipto Guha. Learning to create data-integrating queries. *Proceedings of the VLDB Endowment*, 1(1):785–796, 2008.

[78] Balder ten Cate, Cristina Civili, Evgeny Sherkhonov, and Wang-Chiew Tan. High-level why-not explanations using ontologies. In *Proceedings of the ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 31–43, 2015.

[79] Balder ten Cate, Víctor Dalmau, and Phokion G. Kolaitis. Learning schema mappings. In *International Conference on Database Theory (ICDT)*, pages 182–195, 2012.

[80] Quoc Trung Tran and Chee-Yong Chan. How to conquer why-not questions. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 15–26, 2010.

[81] Marcos Antonio Vaz Salles, Jens-Peter Dittrich, Shant Kirakos Karakashian, Olivier René Girard, and Lukas Blunschi. itrails: pay-as-you-go information integration in dataspaces. In *Proceedings of the 33rd international conference on Very large data bases*, pages 663–674. Proceedings of the VLDB Endowment, 2007.

[82] Jiannan Wang, Tim Kraska, Michael J Franklin, and Jianhua Feng. Crowder: Crowdsourcing entity resolution. *Proceedings of the VLDB Endowment*, 5(11):1483–1494, 2012.

[83] Xiaolan Wang, Xin Luna Dong, and Alexandra Meliou. Data x-ray: A diagnostic tool for data errors. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1231–1245. ACM, 2015.

[84] Mohamed Yakout, Ahmed K. Elmagarmid, Jennifer Neville, Mourad Ouzzani, and Ihab F. Ilyas. Guided data repair. *PVLDB*, 4(5):279–289, 2011.

[85] Zhepeng Yan, Val Tannen, and Zachary Ives. Fine-grained provenance for linear algebra operators. In *8th USENIX Workshop on the Theory and Practice of Provenance (TaPP 16)*, Washington, D.C., June 2016. USENIX Association.

[86] Zhepeng Yan, Nan Zheng, Zachary G Ives, Partha Pratim Talukdar, and Cong Yu. Actively soliciting feedback for query answers in keyword search-based data integration. *Proceedings of the VLDB Endowment*, 6(3):205–216, 2013.

[87] Zhepeng Yan, Nan Zheng, Zachary G Ives, Partha Pratim Talukdar, and Cong Yu. Active learning in keyword search-based data integration. *The VLDB Journal*, pages 1–21, 2015.

[88] Jeffrey Xu Yu, Lu Qin, and Lijun Chang. Keyword search in databases. *Synthesis Lectures on Data Management*, 2009.

[89] Chen Jason Zhang, Lei Chen, HV Jagadish, and Chen Caleb Cao. Reducing uncertainty of schema matching via crowdsourcing. *Proceedings of the VLDB Endowment*, 6(9):757–768, 2013.

[90] Chen Jason Zhang, Yongxin Tong, and Lei Chen. Where to: Crowd-aided path selection. *Proceedings of the VLDB Endowment*, 7(14):2005–2016, 2014.

[91] Dengyong Zhou, Sumit Basu, Yi Mao, and John C Platt. Learning from the wisdom of crowds by minimax entropy. In *Advances in Neural Information Processing Systems*, pages 2195–2203, 2012.