Exploratory Analysis with Imprecise Queries

A dissertation presented
by

Bahar Qarabaqi

to the Faculty of the Graduate School
of the College of Computer and Information Science
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Committee in charge:
Professor Jay Aslam
Dr. Miyoko Chu
Professor Mirek Riedewald, Chair
Professor Yizhou Sun

Northeastern University
Boston, Massachusetts
2015
THESIS TITLE: Exploratory Analysis with Imprecise Queries

AUTHOR: Bahar Farabadi

Ph.D. Thesis Approved to complete all degree requirements for the Ph.D. Degree in Computer Science.

Thesis Advisor: [Signature] 7/30/2015

Thesis Reader: [Signature] 8/1/15

Thesis Reader: [Signature] 8/3/15

Graduate School Approval:

Director, Graduate School: [Signature] 8/25/2015

Copy Received in Graduate School Office:

Recipient's Signature: [Signature] 8/28/2015

Distribution: Once completed, this form should be scanned and attached to the front of the electronic dissertation document (page 1). An electronic version of the document can then be uploaded to the Northeastern University-UMI website.
Very large amounts of data are available in various domains, including scientific research disciplines, online communities and marketing. Unfortunately, the ability to analyze big data and extract interesting information is often limited by the user’s knowledge and expertise in databases. In particular, she has to be able to compose a precise query that specifies exactly what she wants. We have developed Merlin, a new framework for querying big data. Merlin provides new functionality for exploratory search in large databases to find entities of interest even when the user is not sure about some of their properties.

The user interacts with Merlin by specifying probability distributions over attributes, which express imprecise conditions. Merlin helps the user home in on the right query conditions by addressing three key challenges: (1) efficiently computing results for an imprecise query, (2) providing feedback about the sensitivity of the result to changes of individual conditions, and (3) suggesting new conditions. We formally introduce the notion of sensitivity and prove structural properties that
enable efficient algorithms for quantifying the effect of uncertainty in user-specified conditions. To support interactive responses, we also develop techniques that can deliver probability estimates within a given realtime limit and are able to adapt automatically as interactive query refinement proceeds. Due to the interactive nature of the process, query conditions are added incrementally. We show that utilizing a prediction model that is specifically trained for the given query delivers better probability estimates than a model trained on all data attributes. This creates a trade-off between cost of storing models and the quality of predicted probabilities. We show how to overcome this trade-off by training the appropriate models incrementally on-the-fly without violating the response-time requirement.
Acknowledgements

First and foremost, I would like to thank my advisor, Professor Mirek Riedewald, for his mentorship and support during my years at Northeastern University. He is a caring advisor and a brilliant researcher and uses every opportunity to generously share his knowledge and experience with his students. He is inspiring, intelligent, patient and always available for guidance and research discussions. I would not have been able to prepare this dissertation without his advisorship and helpful guidance through every step of my research.

I am grateful to my dissertation committee members, Professor Jay Aslam, Professor Yizhou Sun and Dr. Miyoko Chu for their support and helpful suggestions.

The main part of my work was done in collaboration with the Merlin and eBird teams at the Cornell Lab of Ornithology. I would like to thank everybody involved in the Merlin project and in particular Jessie Barry, Miyoko Chu, Scott Haber and Tim Levatich for providing the data and domain-related advice. I was always inspired by the amazing and enthusiastic bird experts at the Cornell lab of Ornithology.

I thank my former and current lab-mates and colleagues Alper Okcan, Rundong Li and Pavel Metrikov for conducting productive discussions. As the senior student in the Data Mining group, Alper always shared his knowledge and experience with other students. Pavel was also a great help with the technical setup and the paperwork on my defense day.

I would like to thank all my friends in Boston, in particular Hanieh Mirzaei and Sina Khoshosokhan, for always being there for me and for the memories we created.
A special thank goes to Sina for reading and editing parts of this dissertation and for helping me when I needed it the most.

I am very honored to co-found Simorgh Theater Ensemble and Gahnevesht Writing Group in Boston. Every single minute spent in Simorgh and Gahnevesht was amazing and joyful. I would like to thank all former and current members of these groups for the fun we had and the great team work we learned from each other.

Lastly, I would like to thank my family for all their love, support and encouragement. I’m forever grateful to my parents for their selflessness and to my sister, Parastoo, for her inspiring role in my life. This dissertation is dedicated to my family.
Contents

Abstract iii

Acknowledgements v

Contents vii

List of Figures ix

List of Tables xiii

1 Introduction 1
  1.1 Challenges and Contributions . . . . . . . . . . . . . . . . . . . . 5
  1.2 Summary and Dissertation Outline . . . . . . . . . . . . . . . . . 6

2 Imprecise Interactive Queries 7
  2.1 Data Model and Framework . . . . . . . . . . . . . . . . . . . . . . . 7
    2.1.1 Data Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
    2.1.2 Probabilistic Query . . . . . . . . . . . . . . . . . . . . . . . . . 8
    2.1.3 Dealing With Continuous Attributes . . . . . . . . . . . . . . . . 11
    2.1.4 Exploration Framework . . . . . . . . . . . . . . . . . . . . . . . 11
  2.2 Result Ranking . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
  2.3 Sensitivity Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
    2.3.1 Definition and Computation of Sensitivity . . . . . . . . . . . . . . 15
    2.3.2 Sampling Alternative Conditions . . . . . . . . . . . . . . . . . 18
    2.3.3 Ranking Distance Measures . . . . . . . . . . . . . . . . . . . . 24
<table>
<thead>
<tr>
<th>Section Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3.4 Problem Hardness</td>
<td>27</td>
</tr>
<tr>
<td>2.3.5 Collinearity of Probabilities and Impossibility of Repeated Entity Order Swaps</td>
<td>29</td>
</tr>
<tr>
<td>2.3.6 Monotonicity of Ranking Distance</td>
<td>32</td>
</tr>
<tr>
<td>2.4 Recommending Additional Conditions</td>
<td>36</td>
</tr>
<tr>
<td>2.5 Probability Estimation in Realtime</td>
<td>37</td>
</tr>
<tr>
<td>2.5.1 Challenges</td>
<td>39</td>
</tr>
<tr>
<td>2.5.2 Solution: Bagged Tree Ensembles</td>
<td>40</td>
</tr>
<tr>
<td>2.5.3 Controlling Tree Response Time</td>
<td>42</td>
</tr>
<tr>
<td>2.6 Experiments</td>
<td>50</td>
</tr>
<tr>
<td>2.6.1 Data and Models</td>
<td>50</td>
</tr>
<tr>
<td>2.6.2 Sensitivity Analysis</td>
<td>53</td>
</tr>
<tr>
<td>2.6.3 Additional Condition Recommendation</td>
<td>55</td>
</tr>
<tr>
<td>2.6.4 Interactive Response Time</td>
<td>58</td>
</tr>
<tr>
<td>3 Calibration and Prediction Quality</td>
<td>61</td>
</tr>
<tr>
<td>3.1 Calibration Definition</td>
<td>62</td>
</tr>
<tr>
<td>3.2 Calibration with Partial Input</td>
<td>70</td>
</tr>
<tr>
<td>3.2.1 Complete and Reduced-Feature Models</td>
<td>72</td>
</tr>
<tr>
<td>3.2.2 Calibrated Probabilities in Merlin</td>
<td>79</td>
</tr>
<tr>
<td>3.2.3 Layered Models</td>
<td>81</td>
</tr>
<tr>
<td>3.2.4 Experiments</td>
<td>90</td>
</tr>
<tr>
<td>4 Related Work</td>
<td>99</td>
</tr>
<tr>
<td>5 Conclusions and Future Work</td>
<td>107</td>
</tr>
</tbody>
</table>

Bibliography 111
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Possible interface for specifying multiple wing color values</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Possible interface for specifying the bird size</td>
<td>2</td>
</tr>
<tr>
<td>1.3</td>
<td>Feedback to the user after she specified conditions for attributes $X_2$ and $X_4$</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>Attribute $X$ with two possible probability distributions $a_1$ and $a_2$ in $A$</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>Merlin overview. Entity and attribute models are trained from data set $D$. Any model type returning good probabilities can be used, e.g., bagged trees or Bayes nets. At query time, given user-specified conditions, the entity and attribute rankers produce ranked lists like those shown in the center and on the left in Fig. 1.3, respectively. Ranking can be based on different criteria as indicated in the boxes. Using the entity model, the sensitivity analyzer explores how sensitive the current result ranking is to changes the user considers for some attribute's condition.</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>Ranking distance for 3-valued attribute billLength when changing original condition $(0.4, 0.2)$ to another pair $(p_1, p_2)$</td>
<td>16</td>
</tr>
<tr>
<td>2.4</td>
<td>Ranking distance for Boolean attribute hasWingColorBlue when changing original condition $(0.6)$ to another value</td>
<td>17</td>
</tr>
<tr>
<td>2.5</td>
<td>Nonuniform sampling from a 3-simplex</td>
<td>19</td>
</tr>
<tr>
<td>2.6</td>
<td>Uniform sampling from a 3-simplex</td>
<td>19</td>
</tr>
</tbody>
</table>
2.7 The corresponding point on the edge (blue), obtained where the ray connecting the original point (red) to the sampled point (grey) intersects with the edge .................................................. 20
2.8 Sample used by efficient algorithm when the initial point (red) is at 
(1/3, 1/3) .................................................................................. 21
2.9 Sample used by efficient algorithm when the initial point (red) is at 
(0.8, 0.1) .................................................................................. 21
2.10 Local uniform distribution on each edge, due to equal areas of triangles. 
All triangles share the same altitude and their bases have equal length. 22
2.11 To obtain a uniform sample in the entire triangle, sample uniformly in 
small triangles, inversely proportional to their areas $S_1$, $S_2$ and $S_3$. . 22
2.12 Efficient algorithm finds the corresponding point on the edge, no matter 
if the sampled point is inside or outside $\mathcal{A}$. ...................................... 24
2.13 User defines $\mathcal{A}$ by specifying two constraints. The furthest point from 
the original point (red) is not on an edge of $\mathcal{A}$ and is rejected. The 
second point is accepted. ................................................................. 25
2.14 User defines $\mathcal{A}$ by a non-convex space. The furthest point from the 
original point (red) is accepted, because it is on an edge of $\mathcal{A}$. . 26
2.15 For each point $(p_1, p_2)$ the color indicates the Minkowski distance (for 
d = 2) between original ranking for condition $(0.5, 0.4)$ and the ranking 
obtained by changing it to $(p_1, p_2)$. Rankings for each region are 
indicated. ................................................................. 28
2.16 Entity probabilities for collinear conditions $p$, $q$, $r$. .......................... 31
2.17 Possible cases for the ranks of $y$ and $y'$ in $L_\nu$, relative to their position 
in $L_i$ and $L_{i+1}$. ................................................................. 33
2.18 Adaptive Tree restricted to $t \leq T$ models and $l \leq L$ levels . . . . . . . 42
2.19 Tree pruning due to assigning zero probability to some values of an 
attribute ................................................................. 43
2.20 Adaptive Tree keeps track of the number of nodes in the next level
to be able to safely add them whenever desired. The 3 white nodes
are accessed. The 6 light grey nodes are not accessed, but their count
is known. All other nodes in the next levels are not even known to
Merlin. ................................................................. 44

2.21 When user modifies the condition of billLength, the previously pruned
branch is accessed again, but initially with a local level limit of $l_c$. In
the next 2 rounds $l_c$ is gradually increased until it reaches the global
level limit $l$. ................................................................. 46

2.22 Naive vs. efficient sensitivity estimation for a 9-valued attribute . . . 54

2.23 Naive vs. efficient sensitivity estimation for a Boolean attribute . . . 54

2.24 Response time: Imprecision = 0 ........................................ 58

2.25 Response time: Imprecision = 0.5 ....................................... 59

2.26 Response time: Imprecision = 1 ........................................ 59

3.1 One-vs.-rest approach for computing calibration in multi-class prob-
lems ................................................................................. 69

3.2 Calibration graph for full bagged tree ensemble with all known at-
tribute values ................................................................. 71

3.3 An illustration of Top and Low attributes ................................. 73

3.4 Comparison between calibration curves of complete model and
reduced-feature models when 0, 1 and 2 top attributes appearing in
the complete model are selected as the specified attributes. .......... 74

3.5 Comparison between calibration curves of complete model and
reduced-feature models when 4, 8 and 16 top attributes appearing in
the complete model are selected as the specified attributes. .......... 75

3.6 Comparison between calibration curves of complete model and
reduced-feature models when 1, 2 and 4 low attributes are selected as
the specified attributes. ..................................................... 78

3.7 Layered model induction. .................................................. 82
3.8 Calibration curves of layered model when 0, 1, 2, 3, 4 and 5 top attributes appearing in the complete model are selected as the specified attributes. ............................................................ 86
3.9 Calibration curves of layered model when 6, 8, 10, 12, 14 and 16 top attributes appearing in the complete model are selected as the specified attributes. ............................................................ 87
3.10 Calibration curves of layered model when 1, 2, 3 and 4 low attributes are selected as the specified attributes. ............................................................ 89
3.11 Comparison between calibration curves of complete model and reduced-feature models when 0 and 1 top attributes appearing in the complete model are selected as the specified attributes in Adult data set. ............................................................ 91
3.12 Comparison between calibration curves of complete model and reduced-feature models when 2 and 4 top attributes appearing in the complete model are selected as the specified attributes in Adult data set. ............................................................ 92
3.13 Calibration curves of layered model when 0, 1, 2 and 4 top attributes appearing in the complete model are selected as the specified attributes in Adult data set. ............................................................ 93
3.14 Comparison between calibration curves of complete model and reduced-feature models when 0 and 1 low attributes appearing in the complete model are selected as the specified attributes in Adult data set. ............................................................ 95
3.15 Comparison between calibration curves of complete model and reduced-feature models when 2 and 4 low attributes appearing in the complete model are selected as the specified attributes in Adult data set. ............................................................ 96
3.16 Calibration curves of layered model when 0, 1, 2 and 4 low attributes appearing in the complete model are selected as the specified attributes in Adult data set. ............................................................ 97
# List of Tables

2.1 Important notation. .................................................. 8

2.2 List of attributes and their domains in the data set provided by Cornell Lab of Ornithology - Part 1 ........................... 51

2.3 List of attributes and their domains in the data set provided by Cornell Lab of Ornithology - Part 2 ........................... 52

2.4 Comparison of Adaptive Tree (A5 and A1), Full Tree (F) and Random approach (R). The first row shows the initial rank and probability of the species of interest. The next rows show its state after the user added a condition for the attribute recommended by the corresponding method. 56

3.1 Comparison between calibration values of complete model and reduced-feature models when the top attributes appearing in the complete model are selected as the specified attributes. .............. 76

3.2 Comparison between calibration values of complete model and reduced-feature models when four low attributes are selected as the specified attributes. .......................................................... 79

3.3 Comparison between calibration values of complete model, reduced-feature models and layered model when the top attributes appearing in the complete model are selected as the specified attributes. .............. 88

3.4 Comparison between calibration values of complete model, reduced-feature models and layered model when the low attributes are selected as the specified attributes. .............................. 89
3.5 Comparison between calibration values of complete model, reduced-feature models and layered model when the *top* attributes are selected as the specified attributes in Adult data set. . . . . . . . . . . . . . . 93

3.6 Comparison between calibration values of complete model, reduced-feature models and layered model when the *low* attributes are selected as the specified attributes in Adult data set. . . . . . . . . . . . . . . 94
CHAPTER 1

Introduction

We propose Merlin, a new approach for exploratory search in large databases. Merlin provides a novel method to interactively compose imprecise database queries with probabilistic conditions, while providing constant feedback to the user about the most likely results and the potential benefit and risk of each condition. This method is designed to accommodate uncertainty and imprecision in user-provided query conditions through two major technical contributions: (1) a novel notion of sensitivity to quantify the impact of uncertainty on the query result, and (2) fast algorithms for calibrated probability estimation that can adapt to a user-specified realtime constraint on system response time.

To illustrate the need for imprecise queries with probabilistic conditions, consider the following example motivated by a collaboration with the Cornell Lab of Ornithology. Through hugely successful citizen science projects such as eBird (http://ebird.org), the Lab has collected more than 100 million reports of bird sightings, adding tens of millions annually. It wants to leverage this resource to help less experienced birders identify the species of a bird they observed. Assume each observation in the database specifies properties of the bird (e.g., species, size, color) and the observation event (e.g., location, weather, habitat features).\(^1\)

\(^1\)eBird currently does not collect individual bird features such as size and color. The Lab adds those features based on an expert-designed model as discussed in Section 2.6. In the future, successful bird identification could contribute additional records as users already entered the relevant attribute values during the search process.
CHAPTER 1. INTRODUCTION

What color is the wing?
Select 1-3 colors.

Select the approximate size of the bird.
Click on a point on the scale below

Figure 1.1: Possible interface for specifying multiple wing color values

Figure 1.2: Possible interface for specifying the bird size

After observing an individual of an unknown species, the user can then search this database to find species that matched her description.

The main problem with this idea is that the user often is not certain about her query. Consider Amy, who just observed a “small to medium size bird that was mostly blue” in her garden. Through Web forms such as those shown in Figures 1.1 and 1.2, Amy could express such imprecise search conditions. For example, she might decide that the bird was most likely the size of a robin, but is not sure about it. The front-end turns the user input into a probability distribution over the different size values, so that Merlin can work with it. The lower the confidence, the wider the probability mass is distributed.²

After entering the initial search conditions, Merlin can present three types of feedback as shown in Figure 1.3. The table in the center shows the top-ranked species. Amy could go through the list and explicitly eliminate some species or positively identify a likely match. Note that Amy can continue the search even after positive identification, in order to find other likely matches. Alternatively,

²Other interfaces could be used instead, e.g., allowing the user to specify a “score” for each bird size option. The design of the interface and the transformation into a probability distribution are beyond the scope of this dissertation. Merlin works with any probability distribution provided.
User-specified conditions for attributes $X_2$ and $X_4$

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Score</th>
<th>Rank</th>
<th>Species</th>
<th>Specified Attribute</th>
<th>Sensitivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_3$: ShapeGroup</td>
<td>83.27</td>
<td>1</td>
<td>Y245: Eastern Bluebird</td>
<td>$X_2$: Size</td>
<td>206.88</td>
</tr>
<tr>
<td>$X_{14}$: BillLength</td>
<td>81.12</td>
<td>2</td>
<td>Y211: Bluejay</td>
<td>$X_4$: WingColor</td>
<td>18.01</td>
</tr>
<tr>
<td>$X_{11}$: MainColor</td>
<td>74.98</td>
<td>3</td>
<td>Y223: Barn Swallow</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$X_1$: Time</td>
<td>69.47</td>
<td>4</td>
<td>Y212: Western Scrub-Jay</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$X_2$: Location</td>
<td>65.24</td>
<td>5</td>
<td>Y223: Red-breasted Nuthatch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$X_6$: BreastColor</td>
<td>64.02</td>
<td>6</td>
<td>Y242: Blue-grey Gnatcatcher</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$X_6$: BreastPattern</td>
<td>63.18</td>
<td>7</td>
<td>Y223: Tree Swallow</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$X_5$: BackColor</td>
<td>57.79</td>
<td>8</td>
<td>Y331: Indigo Bunting</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$X_{16}$: LegColor</td>
<td>49.35</td>
<td>9</td>
<td>Y246: Western Bluebird</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$X_5$: BellyPattern</td>
<td>48.81</td>
<td>10</td>
<td>Y210: Steller’s Jay</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 1.3: Feedback to the user after she specified conditions for attributes $X_2$ and $X_4$.

she can use the other two tables to refine her query. The table on the left tells her which other attributes would be most helpful in narrowing the search. Since the system does not know what species the user is looking for, an attribute’s potential usefulness in improving result quality is measured based on the intuition that the most useful attributes are those that best separate “winners” from “losers”. For example, breast color would receive a high score if among blue-winged species, some breast colors occur frequently on individuals of some, but not other species. On the other hand, if blue-winged species tend to have mostly the same breast color, then breast color would receive a low score, because it does not help distinguish the species. This is explored in detail in Section 2.4.

The table on the right shows Merlin’s novel sensitivity score, which is specifically designed to support exploratory analysis involving imprecise conditions. Recall Amy’s uncertainty about the bird’s size. Clearly, size is an important feature for distinguishing species. Hence Amy would like to enter it. However, what if she gets it slightly wrong and the correct species is eliminated? Our proposed notion
of sensitivity helps Amy gauge the risk of entering a condition. It quantifies how much the query result (center table in Figure 1.3) could change if Amy were to modify the corresponding condition. Here Merlin could either consider all possible alternative inputs, or let the user specify a range of alternative inputs she considers. (For bird size, Amy could mark lower and upper end of the range of size options she considers in an interface similar to Figure 1.2.) A high sensitivity score might convince Amy to withdraw her input for this attribute and first enter others, e.g., location. She might then try size again at a later time. Since an attribute’s sensitivity depends on the conditions on other attributes, the uncertainty on bird size might have less impact then.

For the system to interact effectively with a user, it needs to respond quickly after the user entered new information. Since the meaning of “interactive” varies depending on the user and application, Merlin lets the user choose her preferred threshold. It will treat it as a realtime constraint, producing a response within the time limit.

Exploratory search with imprecise conditions could benefit many other applications, including product search and online medical advice. For product search, suppose a user wants to leverage the wisdom of the crowd for deciding about a camera purchase. Crowd-sourced camera data will contain a mix of “objective” properties (e.g., megapixels and price) and subjective user evaluations (e.g., if the camera is good for sport photography). In the medical domain, a database of diseases, their symptoms, potential causes (e.g., family history and lifestyle choices), and remedies would similarly be consulted by people not feeling well. As sites like WebMD’s symptom checker (http://symptoms.webmd.com) show, there is great interest in this kind of application. In general, Merlin’s techniques can be applied to any relational database of interest, helping a database user fine-tune imprecise conditions for exploratory analysis.

To handle imprecise query conditions, Merlin relies on a probability-based framework. This raises the challenge of computing accurate probability estimates as the system interacts with the user. Calibration is an evaluation metric used to
1.1. CHALLENGES AND CONTRIBUTIONS

measure the accuracy of predicted probability estimates.

As will become clear later, the interactive query composition process creates a tradeoff between storage cost and calibration quality. Intuitively, materializing more prediction models allows us to choose the better calibrated ones. We explore if it is possible to overcome this tradeoff by training the appropriate models incrementally on-the-fly.

1.1 Challenges and Contributions

We make several contributions aimed at improving support for exploratory search in databases. First, exploratory search usually involves uncertainty; not only in the data [46] but also in the query. To deal with it, we propose a probability-based framework. Notice that for imprecise queries, the result is probabilistic even if the data is precise. Hence, ranking of result records based on their probability is inherent in exploratory search.

Our second contribution helps the user judge the potential risk of specifying a condition she is not certain about. In particular, if getting it just slightly wrong might significantly change the result, then it might be safer to not enter it. To provide this kind of risk-estimation functionality, we introduce the novel notion of sensitivity of a condition and prove structural properties that allow its efficient computation.

Third, while sensitivity quantifies the risk of a condition, additional condition recommendation discusses how to estimate the benefit by identifying the best new conditions to be added in order to improve result quality.

Fourth, as a user-driven process, query refinement should be interactive. Since computation time tends to be high when dealing with large data and imprecise queries, we propose fast approximate estimation techniques that deliver results within a given response time threshold.

Fifth, since it is essential for Merlin to predict well-calibrated probabilities, we study the notion of calibration as an evaluation metric and assess its existing
definitions in order to find strengths and limitations of each.

Finally, we compare calibration of complete models against reduced-feature models and show the trade-off between the quality of probabilities and the training time. We then introduce layered models which are capable of giving well-calibrated probabilities within an interactive time limit.

1.2 Summary and Dissertation Outline

The remainder of this dissertation is organized as follows: In Chapter 2 we introduce Merlin as a framework for exploratory search and present the details of all functionalities supported by Merlin. Next, in Chapter 3 we emphasize the importance of calibration and show the trade-off between having well-calibrated probabilities and the training time and propose layered models as a feasible solution. Chapter 4 covers the related work in the field and identifies the distinctions of our work. We conclude and present some ideas for future work in Chapter 5.
CHAPTER 2

Imprecise Interactive Queries

In this chapter, we propose Merlin as a new approach for exploratory search in large databases. First, we introduce the data model and the probabilistic framework. Then, we explore the details of each functionality supported by Merlin: ranking the results, sensitivity analysis, recommending additional conditions and estimating probabilities in realtime. Finally, we present a proof of concept experimental result for each functionality of Merlin.

2.1 Data Model and Framework

We introduce the data model and propose Merlin’s probabilistic framework for exploratory search in databases. Important notation is summarized in Table 2.1.

2.1.1 Data Model

We are given a relational table or view $D$ with schema $\{X_1, X_2, \ldots, X_m, Y\}$. Attribute $Y$ takes on a special role, identifying entities of interest to the user. Depending on the problem, any attribute of $D$ could take on this role. For instance, in the bird identification example, $Y$ is the species name. When looking for geographical regions, $Y$ would be the corresponding region identifier. Even though $Y$ identifies entities of interest to the user, it does not need to be a key of $D$. The tuples in $D$ could represent precise or probabilistic information, including crowd-sourced imprecise data.
CHAPTER 2. IMPRECISE INTERACTIVE QUERIES

Table 2.1: Important notation.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D$</td>
<td>given relational table or view with schema ${X_1, X_2, \ldots, X_m, Y}$</td>
</tr>
<tr>
<td>$X$</td>
<td>data attribute, e.g., hasWingColorRed with domain ${Y, N}$, for which the user can specify a condition in the query</td>
</tr>
<tr>
<td>$Y$</td>
<td>data attribute identifying entities of interest, e.g., species of a bird</td>
</tr>
<tr>
<td>$A$</td>
<td>set of all possible probability distributions over the values in the domain of attribute $X$, e.g., ${(p_1, p_2) \mid p_1, p_2 \geq 0, p_1 + p_2 = 1}$ for hasWingColorRed</td>
</tr>
<tr>
<td>$a \in A$</td>
<td>specific probability distribution over the values in the domain of attribute $X$, e.g., $(0.2, 0.8)$</td>
</tr>
<tr>
<td>$k$</td>
<td>number of attributes for which the user has specified conditions</td>
</tr>
<tr>
<td>$\Pr(Y \mid A_1, \ldots, A_k, \bar{y}_1, \ldots, \bar{y}_l, D)$</td>
<td>entity probability, given distribution $a_i \in A_i$ for attribute $X_i$, $i = 1, \ldots, k$, and explicit rejection of entity $y_j$, $j = 1, \ldots, l$</td>
</tr>
<tr>
<td>$\Pr(A = a \mid A_1, \ldots, A_k, \bar{y}_1, \ldots, \bar{y}_l, D)$</td>
<td>probability that the user will specify condition $a$ for attribute $X$, given distribution $a_i \in A_i$ for attribute $X_i$, $i = 1, \ldots, k$, and explicit rejection of entity $y_j$, $j = 1, \ldots, l$</td>
</tr>
<tr>
<td>$\phi_j$</td>
<td>effort required from the user to decide if entity $y_j \in Y$ is of interest; e.g., measured as user response time after the entity is presented</td>
</tr>
<tr>
<td>$L_p$</td>
<td>ranked list of entities based on user-specified condition $p$</td>
</tr>
<tr>
<td>$\rho_p(y)$</td>
<td>rank of entity $y$ in ranked list $L_p$</td>
</tr>
<tr>
<td>$A_i \subseteq A_i$</td>
<td>alternative conditions the user considers for attribute $X_i$</td>
</tr>
</tbody>
</table>

In the bird example, the entities of interest are bird species, e.g. Tree Swallow and American Robin. The $X_i$ describe various properties of a bird and observation event. For instance, hasWingColorRed is a Boolean attribute, describing whether the color red is observed on the wing of the bird. Another example is obsLongitude which is a numerical attribute, showing the longitude of the observation location. A species will be observed more than once, hence there will be multiple records for it. For example, many bird enthusiasts observe and report Blue Jays. Since not all individuals of a species look alike or are seen at the same location, the values of the $X_i$ can differ even for records with the same $Y$-value. (For this reason $Y$ is not necessarily a key of $D$.)

2.1.2 Probabilistic Query

The user wants to find entities $y \in Y$ of interest (we slightly abuse notation and use $Y$ to denote both the name of the attribute and its domain) and expresses her preferences by specifying conditions on some of the attributes $X_i$. In a traditional
relational database setting, the user would then execute query

```
SELECT Y, COUNT(*) AS freq
FROM D
WHERE condition(X_1) AND ... AND condition(X_m)
    AND Y <> y_1 AND ... AND Y <> y_l
GROUP BY Y
ORDER BY freq DESC
```

to determine which of the entities are most frequently associated with attribute values satisfying the specified conditions. The second part of the WHERE clause reflects explicit rejection of entities \( y_1, y_2, \ldots, y_l \). If we divide the freq attribute of each result tuple by the total count of tuples in \( D \) satisfying the WHERE clause, then we obtain for each \( y \in Y \) the fraction it represents in the result. More generally, this query would approximate probability \( \Pr(Y = y \mid \text{condition}(X_1), \ldots, \text{condition}(X_m), Y \neq y_1, \ldots, Y \neq y_l, D) \).

To accommodate uncertainty, the WHERE clause needs to support probabilistic conditions. For some attribute \( X_i \), let \( A_i \) be the set of all possible probability distributions over the values in the domain of \( X_i \), e.g., \( \{(p_1, p_2) \mid p_1, p_2 \geq 0, p_1 + p_2 = 1\} \) for Boolean attribute hasWingColorRed with domain \( \{Y, N\} \), or similarly \( \{(p_1, p_2, \ldots, p_9) \mid \forall i : p_i \geq 0, \sum_{i=1}^{9} p_i = 1\} \) for 9-valued attribute size with domain \( \{1, 2, \ldots, 9\} \). When interacting with Merlin, the user specifies some probability distribution \( a_i \in A_i \), e.g., \((0.2, 0.8)\) for hasWingColorRed. Figure 2.1 illustrates an attribute \( X \): size (Figure 2.1a) with two possible probability distributions \( a_1 \) and \( a_2 \) in \( A \) that the user may specify for this attribute. (Figure 2.1b and Figure 2.1c)

Given user-specified conditions \( a_1, \ldots, a_k \) for attributes \( X_1, \ldots, X_k \), the probability of entity \( y \in Y \) is defined as \( \Pr(Y = y \mid A_1 = a_1, \ldots, A_k = a_k, Y \neq y_1, \ldots, Y \neq y_l, D) \), written more compactly as

\[
\Pr(Y \mid A_1, \ldots, A_k, \bar{y}_1, \ldots, \bar{y}_l, D).
\]

Here \( \bar{y}_j \) indicates that entity \( y_j \) was rejected by the user. When looking for multiple results, the user would not only reject entities, but could also accept some and
Figure 2.1: Attribute \( X \) with two possible probability distributions \( a_1 \) and \( a_2 \) in \( A \) then continue the search process. The probabilities can be adjusted accordingly by removing the accepted entities from consideration.

Probability distributions provide flexibility for expressing a mix of certain and uncertain conditions. Recall user Amy who is not certain about the observed bird’s size. She might know for sure that the bird was larger than a sparrow and smaller than a crow, but cannot decide between the three size options around the American Robin (see Figure 1.2). She can express this by entering a distribution like \((0, 0, 0.2, 0.6, 0.2, 0, 0, 0, 0)\) for the nine possible values of the size attribute. The six zeros indicate her certainty about ruling out the overly small and overly large size options. Sensitivity analysis (Section 2.3) helps Amy determine if a small change of the non-zero values would have a major impact on the species ranking.

It is conceptually easy to extend Merlin to support an inequality condition such as \((\text{size} \geq 2 \ \text{AND} \ \text{size} \leq 4)\) which specifies a range of values for size. Instead of concrete distribution \((0, 0, 0.2, 0.6, 0.2, 0, 0, 0, 0)\), the user could specify a set of distributions \((0, 0, q_1, q_2, q_3, 0, 0, 0, 0)\), subject to \(0 \leq q_1, q_2, q_3\) and \(q_1 + q_2 + q_3 = 1\).
Then the formula for the probability of $Y$ becomes an expectation over the different concrete distributions in this set. This is similar to the way we deal with possible future attribute conditions (Section 2.4).

### 2.1.3 Dealing With Continuous Attributes

In the remainder of this dissertation, we assume that all attributes of $D$ have a discrete domain. In principle, Merlin could be extended to also support continuous domains. However, this additional complication does not provide any benefits for exploratory search. Recall that the user provides the distributions as conditions for her query. From the user’s point of view, it makes virtually no sense to try and distinguish between distributions like $(0.8, 0.2)$ and $(0.801, 0.199)$. They both express that the user was highly confident about one of the attribute’s values. And as our analytical results for sensitivity show (Section 2.3), entity probabilities are “well-behaved” in the sense that a smaller change in distribution results in a smaller change in entity probability. Similarly, if an attribute has a very large domain, it can be compressed by representing a distribution over this domain with an appropriate histogram. In general, Merlin can approximate a continuous distribution as closely as desired.

### 2.1.4 Exploration Framework

Figure 2.2 shows the Merlin system components and their interactions. From the given data set $D$, two types of models are trained at “setup time”, i.e., before the system is available for user queries. The entity model predicts $\Pr(Y \mid A_1, \ldots, A_k, \bar{y}_1, \ldots, \bar{y}_l, D)$, i.e., in the bird example the probability of each species. The attribute model predicts the probability of a distribution the user might enter if prompted by Merlin to provide a condition on another attribute such as head color. The probabilities output by these models are used at “query time” by the ranking functions to produce the lists in the center and on the left, respectively, in Figure 1.3. The sensitivity analyzer also relies on the entity model in order to
CHAPTER 2. IMPRECISE INTERACTIVE QUERIES

Figure 2.2: Merlin overview. Entity and attribute models are trained from data set $D$. Any model type returning good probabilities can be used, e.g., bagged trees or Bayes nets. At query time, given user-specified conditions, the entity and attribute rankers produce ranked lists like those shown in the center and on the left in Fig. 1.3, respectively. Ranking can be based on different criteria as indicated in the boxes. Using the entity model, the sensitivity analyzer explores how sensitive the current result ranking is to changes the user considers for some attribute’s condition.

determine how much a modification of one of the user-specified conditions would affect the entity ranking. Details are discussed in the following sections.

2.2 Result Ranking

Given user-specified conditions on the attributes of $D$, Merlin creates a ranked list of the entities $y \in Y$ as shown in the center of Figure 1.3. We call this component the entity ranker or eRank. The entity ranker could directly rank by entity probability $Pr(Y \mid A_1, \ldots, A_k, \bar{y}_1, \ldots, \bar{y}_l, D)$. We propose a measure that generalizes ranking by entity probability by taking user effort into account. It is motivated by the fact that the user has to invest some time to look at the presented query result in order to decide which of the top-ranked entities might be of interest. In the exam-
2.2. RESULT RANKING

In Figure 1.3, the user scans the list of species in the center from top to bottom. Let \( \phi_j \) denote the user effort required for deciding about the relevance of entity \( y_j \).\(^1\) Now consider two entities \( y_1 \) and \( y_2 \) with probabilities 0.51 and 0.49 and effort \( \phi_1 = 10 \) and \( \phi_2 = 1 \), respectively. If we rank \( y_1 \) before \( y_2 \), the user would have to invest an expected effort of \( 0.51 \cdot 10 + 0.49 \cdot (10 + 1) = 10.49 \) when examining the ranked list top-down until the correct answer is found (assuming either \( y_1 \) or \( y_2 \) is correct, but not both). If \( y_2 \) was ranked first, expected effort would drop to \( 0.49 \cdot 1 + 0.51 \cdot (1 + 10) = 6.1 \).

This example motivates **ranking by effort-adjusted entity probabilities**. The *effort-adjusted probability* of entity \( y_i \in Y \) is defined as \( \Pr(Y = y_i \mid A_1, \ldots, A_k, \bar{y}_1, \ldots, \bar{y}_l, D) / \phi_i \). Effort-adjusted probability has the following useful property:

**Lemma 2.2.1.** Assume the user is looking for a single entity of interest by exploring the ranked list of entities one-by-one from top to bottom, until this entity is found. Expected user effort then is minimized if the entities are ranked in decreasing order of their effort-adjusted probability.

**Proof.** Without loss of generality, assume the entities are ranked in order \( y_1, y_2, \ldots, y_n \). If \( y_c \) is the correct result, the user would have to go through entities \( y_1 \) to \( y_{c-1} \), which are all ranked higher, until finding \( y_c \). The corresponding total effort is \( \sum_{j=1}^{c-1} \phi_j \). Hence, we obtain the expected user effort as

\[
\sum_{i=1}^{n} \Pr(Y = y_i \mid A_1, \ldots, A_k, \bar{y}_1, \ldots, \bar{y}_l, D) \cdot \sum_{j=1}^{i} \phi_j.
\]

\( (2.1) \)

Assume for contradiction that ranking \( y_1, y_2, \ldots, y_n \) minimizes expected user effort, but entities are *not* ranked in decreasing order of their effort-adjusted probabilities. Then there exists a pair of adjacent (in the ranking) entities \((y_k, y_{k+1})\) that is not correctly ranked by effort-adjusted probability, i.e., \( P_k / \phi_k < P_{k+1} / \phi_{k+1} \). (To avoid clutter, we use \( P_i \) as a shorthand for \( \Pr(Y = y_i \mid A_1, \ldots, A_k, \bar{y}_1, \ldots, \bar{y}_l, D) \).)

\(^1\)Effort can vary, e.g., some bird species is easily recognizable from a picture while another requires reading a description. In practice effort can be measured based on the user’s response time when interacting with Merlin.
We show that swapping their positions will decrease expected effort, contradicting the assumption of minimality.

It is easy to see that swapping $y_k$ and $y_{k+1}$ in Eq. 2.1 changes expected user effort by $P_k\phi_{k+1} - P_{k+1}\phi_k$. This difference is negative because of $P_k/\phi_k < P_{k+1}/\phi_{k+1}$. Hence the initial ranking $y_1, y_2, \ldots, y_n$ did not minimize expected user effort, completing the proof of Lemma 2.2.1 by contradiction.

\[\square\]

2.3 Sensitivity Analysis

When dealing with imprecise queries, it is essential to give the user feedback about the potential risk of a condition. We propose a very direct and natural measure that quantifies the sensitivity of the query result to changes in the condition. Intuitively, user-specified condition $a$ for attribute $X$ has high sensitivity if a “small” change of the condition would result in a “large” change of the entity ranking.

To illustrate the usage of sensitivity, we return to the bird identification example. Assume Amy entered condition $(0.4, 0.2, 0.4)$ for billLength, which is a 3-valued attribute with domain \{1, 2, 3\}. Since she is not really sure about the exact probabilities, she asks Merlin how much the current species ranking could possibly change if she were to enter any other probability distribution $(p_1, p_2, p_3)$ instead. If Merlin determines that the resulting change could be “large”, she might decide to remove her input for billLength. Alternatively, assume she is almost sure that the billLength value should be 1 or 3, but cannot decide how much probability mass exactly to assign. In particular, she considers all distributions $(p_1, p_2, p_3)$ that satisfy $p_1 \geq 0.4 \land p_3 \geq 0.4$. If Merlin tells her that sensitivity in that case is “low”, she knows that no matter if she enters $(0.4, 0.2, 0.4)$ or $(0.5, 0.1, 0.4), (0.4, 0.15, 0.45)$ etc., the ranking would be similar. Notice that this does not mean that the condition on billLength is irrelevant. The ranking might change significantly if Amy chooses a condition that does not satisfy $p_1 \geq 0.4 \land p_3 \geq 0.4$. Low sensitivity simply gives her confidence that the exact choice of values does not matter much in the range she considers.
As will become clearer soon, sensitivity of an attribute depends on the conditions on other attributes. Hence an attribute with high sensitivity might have much lower sensitivity later on, after conditions on other attributes are modified.

### 2.3.1 Definition and Computation of Sensitivity

**Definition 2.3.1.** Let $L$ be the entity ranking based on user-specified conditions $a_1, \ldots, a_k, \bar{y}_1, \ldots, \bar{y}_l$, and let $A_i$ denote the set of all alternative conditions the user considers for some attribute $X_i$, $i \in \{1, \ldots, k\}$. The sensitivity of the current ranking $L$ to attribute $X_i$ for a set of possible conditions $A_i$ is defined as the maximum difference $\text{dst}(L, L')$ between $L$ and any other ranking $L'$ that could be obtained if the user were to change $X_i$’s condition $a_i$ to any other distribution $a_i' \in A_i$. Function $\text{dst}()$ is a distance measure between two entity rankings.

For brevity, we will refer to the “sensitivity of the current ranking $L$ to attribute $X_i$ for a set of possible conditions $A_i$” simply as the *sensitivity of attribute $X_i$*. Note that the sensitivity of $X_i$ is affected by the other conditions specified, because all conditions together determine the entity probabilities and hence the ranking. The choice of $A_i$ also plays a crucial role. Constraints on alternative conditions considered can be specified by eliminating values from the domain of attribute $X_i$ or by limiting the range of probabilities considered for each value. The above example illustrated this for billLength. Similarly, distributions near the specified one can be considered by setting $A_i$ to include an $\varepsilon$-neighborhood. For billLength with condition $(0.4, 0.2, 0.4)$, this could be defined as the set of probability distributions $(p_1, p_2, p_3)$ that satisfy $|p_1 - 0.4| \leq \varepsilon \land |p_2 - 0.2| \leq \varepsilon \land |p_3 - 0.4| \leq \varepsilon$.

Figure 2.3 shows sensitivity for billLength measured in a bird observation data set (discussed in Section 2.6). Notice that any probability distribution $(p_1, p_2, p_3)$ for a 3-valued attribute is uniquely determined by $(p_1, p_2)$, because $p_3 = 1 - p_1 - p_2$. Hence we only need to consider distributions over the first two values. Starting with the original user-specified condition $p = (0.4, 0.2)$, we sampled 1 million alternative conditions randomly and computed the Minkowski distance (see Section 2.3.3).
in ranking to $L$. As the colors indicate, distances increase as one “moves away” from $(0.4, 0.2)$. In this example, the greatest distance is obtained in the lower right corner, for distribution $(1.0, 0.0)$. However, near $(0.4, 0.2)$, distance is comparably low, indicating that if the user considers a small neighborhood around the original condition, then the ranking is stable. Figure 2.4 shows sensitivity for Boolean attribute hasWingColorBlue and initial condition 0.6. (To put the numbers into perspective, note that if each of the 367 species is 5 positions off, total ranking difference would be about 100.)

One can compute the sensitivity of attribute $X_i$ using the following **naive algorithm**: Let $L$ denote the entity ranking based on the user-specified conditions $a_1, \ldots, a_k, \bar{y}_1, \ldots, \bar{y}_l$. Repeat the following: Select a condition $a_i'$ from $A_i$ and compute the entity ranking $L'$ based on the modified condition where $a_i$ is replaced...
2.3. SENSITIVITY ANALYSIS

Figure 2.4: Ranking distance for Boolean attribute hasWingColorBlue when changing original condition (0.6) to another value

by \( a'_i \). Keep track of the maximum distance between rankings \( L \) and \( L' \) and return it to the user.

For billLength, the naive algorithm has to sample from the entire colored triangular area shown in Figure 2.3. Fortunately, as we prove below, this is not necessary. Ranking distance increases monotonically as one conceptually moves on a line away from the initial condition, i.e., \((0.4, 0.2)\) in the example. (There is some “color noise” in the graph, which are artifacts of the drawing process when interpolating between sample points.) Because of this monotonicity property, it is guaranteed that the greatest distance will be found on the edge of the space of possible conditions, i.e., the three sides of the triangle in Figure 2.3. Hence instead of naively sampling from the entire triangle, we can use a more efficient algorithm that samples only from its sides, which dramatically reduces the search space. Similarly, if the user constrains the space of distributions, e.g., by “cutting off” the top
corner of the triangle with constraint $p_2 \leq 0.3$, Merlin only has to sample from the edges of the resulting shape, not from its inner points. The implications are particularly powerful for Boolean attributes. Because ranking distance increases monotonically toward the extremes of the range (as Figure 2.4 shows for the bird data), to find the condition that results in the greatest ranking difference, one only needs to check the rankings for the lower and upper extreme of the range of possible probability values considered.

### 2.3.2 Sampling Alternative Conditions

In order to compute sensitivity, we need to sample random points uniformly from the set of alternative conditions considered, $A$. If $A$ is the entire space of possible probability distributions for the attribute examined, it forms a $k$-simplex where $k$ is the domain size of the attribute. The naive algorithm samples from the interior points, while the points sampled by the efficient algorithm are on the edges of the simplex.

#### 2.3.2.1 Sampling Uniformly from $k$-Simplex

Drawing a uniform sample for a Boolean attribute is easy: pick $p_1$ uniformly at random from $[0, 1]$ and let $p_2 = 1 - p_1$. When generalizing to a $k$-simplex, an intuitive algorithm is to select a sequence $p_1, p_2, \ldots, p_k$, each uniformly at random from $[0, 1]$ and normalize them, so that $\sum_{i=1}^{k} p_i = 1$. But a simple experiment shows that this sampling algorithm (as well as many other similar ones) does not generate a uniform sample. Using this algorithm, we sampled 10000 points from a 3-simplex and plotted the points in two dimensional space. The third dimension can be derived from the other two. Figure 2.5 provides obvious evidence that this sample is not uniform.

Obtaining a uniform sample from the $k$-simplex is a special case of sampling from a Dirichlet distribution with parameters $\alpha_1, \ldots, \alpha_k$ all set to 1 [43]. The algorithm to generate a uniform sample from the $k$-simplex using Dirichlet distribution
2.3. SENSITIVITY ANALYSIS

is as follows:

1. Draw \( k \) points \( a_i \) for \( i \in \{1, \ldots, k\} \) from \([0, 1]\) uniformly.

2. For each \( i \), compute \( b_i = -\log a_i \)

3. To normalize the result, compute the sum \( s = \sum_{i=1}^{k} b_i \)

4. Normalize the values to get \( p_i = \frac{b_i}{s} \)

Figure 2.6 shows the same experiment for this algorithm.

2.3.2.2 Sampling on Edge

We showed how to sample uniformly from a simplex. This uniform sample is directly used by the naive algorithm as the set of alternative user responses. One option for the efficient algorithm is to take each point in this set and return its corresponding point on the edge, obtained where the ray connecting the original point to the sampled point intersects with the edge. Figure 2.7 shows the ray connecting the original point (in red) to the sampled point (in grey) which is continued until it intersects with the edge of the triangle and the corresponding point on the edge (in blue) is achieved.
This approach guarantees that after exploring the \( i \)-th sample, the intermediate maximum distance found by the efficient algorithm is greater than or equal to the maximum distance found by the naive algorithm. We use this approach in our experiments. Note that the final points used by the efficient algorithm are not necessarily uniformly distributed on the edges of the simplex. Figure 2.8 illustrates a sample of 500 points obtained using the connecting ray approach when the initial point has equal values in all dimensions. The small number of samples is due to illustration purposes. If we sample more points, they form a line and their distribution is no longer visible. In the special case of Figure 2.8, the points on the edge have a uniform distribution. But if we move the initial point closer to one of the
2.3. SENSITIVITY ANALYSIS

Figure 2.8: Sample used by efficient algorithm when the initial point (red) is at \((1/3, 1/3)\)

Figure 2.9: Sample used by efficient algorithm when the initial point (red) is at \((0.8, 0.1)\)

edges, the distribution is no longer uniform. Figure 2.9 shows the set of alternative points when the initial point is at \((0.8, 0.1)\).

No matter where the initial point is, we show that the distribution of points on each edge is indeed uniform. The intuition behind this observation is illustrated in Figure 2.10. Divide the \(p_1\) axis into equal-length line segments of length \(w\). Connect the two end points of each line segment to the initial point to form a triangle. The areas of all these triangles are equal, since the height \(h\) and the length \(w\) of the base is the same for all of them. Since samples are distributed uniformly in the simplex, each of the small triangles has the same probability of containing a given sample point. This in turn implies that each of the segments of length \(w\) has the same probability of containing the corresponding projected point. This holds for any value of \(w\) and hence, the sample distribution on each edge of the triangle is uniform. However, the distribution is not necessarily the same across different edges.

This gives the intuition for the second sampling option for the efficient algorithm which achieves a uniform sampling along all edges, but no longer guarantees to find a maximum distance that is not less than the maximum distance found by the naive algorithm. This unfortunate case happens if the naive algorithm draws a
Figure 2.10: Local uniform distribution on each edge, due to equal areas of triangles. All triangles share the same altitude and their bases have equal length.

Figure 2.11: To obtain a uniform sample in the entire triangle, sample uniformly in small triangles, inversely proportional to their areas $S_1$, $S_2$ and $S_3$.

lucky sample point on or close to the edge, having the maximum distance. At the same time, the efficient algorithm is unlucky and does not sample any point from that part of the edge.

To draw a uniform sample along all edges, consider the triangles obtained from connecting the initial point to the vertices of the original triangle, as shown in Figure 2.11. First select one of the three regions with probability inversely proportional to the corresponding area $S_1$, $S_2$ and $S_3$. Then sample uniformly from this region. This idea generalizes to higher dimensions.

### 2.3.2.3 Sampling from Arbitrary $\mathcal{A}$

If the user restricts the set of alternative responses, $\mathcal{A}$, then a subset of points in the $k$-simplex must be considered. This complicates the sampling process. In this case and for an arbitrary shape of $\mathcal{A}$, the naive algorithm follows a rejection strategy: sample a point in the $k$-simplex and check if it is in $\mathcal{A}$. If so, accept the point. Otherwise, reject it. If the volume of $\mathcal{A}$ is very small compared to the entire volume of the simplex, the chance of a hit decreases and more and more samples must be
drawn in order to obtain one accepted sample.

The efficient algorithm does not suffer from low hit probability even for small $A$s. Recall that the efficient algorithm does not directly sample from the edges of the simplex, but samples from the entire simplex and returns the point on the edge, obtained where the ray connecting the original point to the sampled point intersects with the edge. This approach does not depend on the relative position of the sampled point and $A$. Figure 2.12 illustrates how to find a point on the edge, no matter where the sampled point is located. If the sampled point is inside $A$, we connect the original point and the sampled point and continue until the ray intersects with $A$. If the sampled point is outside $A$, the desired point would be between the original point and the sampled point.

Note that we always consider a directed ray from the original point to the sampled point to obtain a single point on the edge. But if the user defines $A$ by specifying more than one constraint, as in Figure 2.13, or by an arbitrary non-convex space, as in Figure 2.14, the connecting ray might still intersect more than once with the edges of $A$. In that case, following Theorem 2.3.4, we consider the furthest point satisfying all constraints. Figure 2.13 illustrates the case where the furthest point which is considered first does not satisfy all constraints and hence, is rejected. The second point is accepted, because it satisfies both constraints and is indeed on the edge of $A$. Similarly, Figure 2.14 shows a ray with multiple intersections. This time, the furthest point is already on the edge of $A$ and is accepted.

Next we will introduce ranking distance measures and prove the monotonicity property that implies that it is sufficient to sample from the edges. We also provide evidence that it is unlikely that a more efficient general algorithm exists by showing that the problem is not convex and that the optimal solution does not necessarily lie in a vertex.
2.3.3 Ranking Distance Measures

Merlin’s sensitivity analyzer can work with any distance measure $\text{dst}()$ between entity rankings. We introduce three important examples and show in Section 2.3.6 that they all admit an optimization algorithm that is much more efficient than the naive algorithm. The first two correspond to (generalizations of) the two most commonly used ranking correlation measures in information retrieval [50]. Let $L_p$ and $L_q$ be two rankings of the $n$ entities in $Y$. For each entity $y \in Y$, let $\rho_p(y)$ and $\rho_q(y)$ denote $y$’s rank in the corresponding ranking.
2.3. SENSITIVITY ANALYSIS

Figure 2.13: User defines \( A \) by specifying two constraints. The furthest point from the original point (red) is not on an edge of \( A \) and is rejected. The second point is accepted.

**Minkowski distance**: The distance of two rankings can be measured based on the difference in rank for each individual entity. For instance, the Spearman rank correlation coefficient, which is widely used for evaluating similarity between rankings in information retrieval, relies on the squared rank differences and is defined as

\[
1 - \frac{6 \sum_{y \in Y} (\rho_p(y) - \rho_q(y))^2}{n(n^2 - 1)}
\]

We consider not only squared differences, but the **Minkowski distance** in general. For some constant \( d > 0 \) it is defined as

\[
\text{dst}(L_p, L_q) = \left( \sum_{y \in Y} |\rho_p(y) - \rho_q(y)|^d \right)^{1/d}.
\]

(2.2)
Figure 2.14: User defines $A$ by a non-convex space. The furthest point from the original point (red) is accepted, because it is on an edge of $A$.

**Kendall’s τ distance:** Kendall’s $\tau$ is another commonly used measure of correlation between two rankings. It is defined as $\frac{C-D}{n(n-1)/2}$, where $C$ and $D$ denote the total number of entity pairs that are ranked in the same and opposite order, respectively, in the two rankings. Finding the maximal ranking distance is equivalent to finding the minimal correlation. Since $n$, the number of different entities, is a constant for a given problem, and $C + D = n(n+1)/2$, Kendall’s $\tau$ increases linearly with $-D$. Hence we define a ranking distance measure based on Kendall’s $\tau$ as

$$\text{dst}_\tau(L_p, L_\eta) = D.$$  \hspace{1cm} (2.3)
Intuitively, it measures the number of inversions between the two rankings.

**Average Precision distance**: Minkowski distance and Kendall’s $\tau$ weigh ranking differences equally, no matter if they occur near the top or near the bottom of the ranking. Recent work in the information retrieval community proposed the Average Precision (AP) correlation coefficient $\tau_{\text{AP}}$ [50] as a measure that weighs differences near the top higher, because those are the differences that more likely affect a user during Web search. It is based on $C_{p,q}(k)$, which measures for entity $y$ at position $k$ in ranking $L_q$, how many entities among those above it in $L_q$ are also ranked above $y$ in the other ranking $L_p$. Formally,

$$
\tau_{\text{AP}}(L_p, L_q) = \frac{2}{n-1} \sum_{k=2}^{n} \frac{C_{p,q}(k)}{k-1} - 1.
$$

Finding the maximal ranking distance is equivalent to finding the minimal correlation, hence we use the negative AP correlation coefficient as a the corresponding distance measure

$$
dst_{\text{AP}}(L_p, L_q) = -\tau_{\text{AP}}(L_p, L_q).
$$

### 2.3.4 Problem Hardness

We show that sensitivity computation in general is not a convex optimization problem and that the optimal solution might not lie in a vertex.

**Convexity**: If the set of alternative probability distributions, $A_i$, is not convex, then the optimization problem is not convex. Even for convex $A_i$, the objective function is not necessarily convex as shown by a simple counter-example.

Let $X$ be an attribute with domain $\{x_1, x_2, x_3\}$ and let $a_1 = (1, 0, 0)$, $a_2 = (0, 1, 0)$ and $a_3 = (0, 0, 1)$ be probability distributions over this domain. Assume there are three entities $y_1$, $y_2$ and $y_3$ with the following entity probabilities (data $D$ is omitted in the formulas for brevity): $\Pr(Y = y_1 | A = a_1) = 0.5$, $\Pr(Y = y_1 | A = a_2) = 0.4$, $\Pr(Y = y_1 | A = a_3) = 0.1$, $\Pr(Y = y_2 | A = a_1) = 0.4$, $\Pr(Y = y_2 | A = a_2) = 0.1$, $\Pr(Y = y_2 | A = a_3) = 0.4$, $\Pr(Y = y_3 | A = a_1) = 0.1$, $\Pr(Y = y_3 | A = a_2) = 0.5$, $\Pr(Y = y_3 | A = a_3) = 0.5$. 
Let the original condition for $X$ be $a = (0.5, 0.4, 0.1)$, and let $\mathcal{A} = \{(p_1, p_2) \mid p_1, p_2 \geq 0, p_1 + p_2 \leq 1\}$. (This considers any probability distribution over the domain of $X$.) We randomly sampled 1.5 million distributions $a' = (p_1, p_2)$ from $\mathcal{A}$ and computed the Minkowski distance, using squared rank differences, of the resulting entity ranking to the ranking for the original distribution $a$. Figure 2.15 shows the result for all combinations of $p_1$ (x-axis) and $p_2$ (y-axis). (Recall that $p_3 = 1 - p_1 - p_2$.) The ranked list for original condition $a$ is $(y_1, y_3, y_2)$. In the bright red region, the distance is $\sqrt{6}$ and in the dark red region it is $\sqrt{8}$. Because of the discontinuity at the boundary between regions, it is easy to show that the objective function is neither convex nor concave.

**Optimum in a vertex:** If the maximum ranking distance could be shown to lie in one of the corners of the triangle, only those three rankings would have to be
2.3. **SENSITIVITY ANALYSIS**

explored when computing sensitivity. Unfortunately, in the example the distance between the original ranking and those obtained in the three corners of the triangle are $\sqrt{2}$, $\sqrt{2}$ and $\sqrt{6}$, respectively. For $\alpha' = (0.4, 0)$ in the dark red region, the ranked list is $(y_2, y_3, y_1)$, resulting in a distance of $\sqrt{8}$ to the original ranking.

This counter-example can also be used to show the same negative results for the other two ranking distance measures, Kendall’s $\tau$ distance and Average Precision distance. Fortunately, as discussed next, we are able to prove the “next best” structural property: that the optimum always lies on the “edge” of the set of possible conditions $A$, i.e., the sides of the triangle in the example.

### 2.3.5 Collinearity of Probabilities and Impossibility of Repeated Entity Order Swaps

The proof that the optimum lies on the “edge” of the set of possible conditions $A$ relies on two general properties, stated in Lemmas 2.3.2 and 2.3.3, which hold independent of the ranking distance measure.

**Lemma 2.3.2.** Let $p$, $r$, and $q$ be probability distributions for attribute $X$, and let the corresponding entity probabilities be $P = \Pr(Y \mid A = p, A_1, \ldots, A_k, \bar{y}_1, \ldots, \bar{y}_l, D)$, $Q = \Pr(Y \mid A = q, A_1, \ldots, A_k, \bar{y}_1, \ldots, \bar{y}_l, D)$, and $R = \Pr(Y \mid A = r, A_1, \ldots, A_k, \bar{y}_1, \ldots, \bar{y}_l, D)$. If $q = p + \alpha \cdot (r - p)$, $0 < \alpha < 1$, then $Q = P + \alpha \cdot (R - P)$.

**Proof:** For simplicity and without loss of generality, collinearity will be shown for an attribute $X$ with three possible values $x_1$, $x_2$, and $x_3$. Hence a condition on $X$ is a probability distribution $p \in \{(p_1, p_2, p_3) \mid p_1, p_2, p_3 \geq 0, p_1 + p_2 + p_3 = 1\}$. Since the third probability is determined by the other two (all have to add up to 1), we only need to consider a probability vector $(p_1, p_2)$.

Consider three different conditions for attribute $X$, expressed as probability vectors $p = (p_1, p_2)$, $q = (q_1, q_2)$, and $r = (r_1, r_2)$. Let the three vectors satisfy $q = p + \alpha \cdot (r - p)$ for some $0 < \alpha < 1$, i.e., $p$, $q$, and $r$ are collinear conditions.
for attribute $X$. (Intuitively, $q$ lies on the line connecting $p$ and $r$.) We show that the corresponding entity probabilities

\[\begin{align*}
P &= \Pr(Y \mid A = p, A_1, \ldots, A_k, \bar{y}_1, \ldots, \bar{y}_t, D) \\
Q &= \Pr(Y \mid A = q, A_1, \ldots, A_k, \bar{y}_1, \ldots, \bar{y}_t, D) \\
R &= \Pr(Y \mid A = r, A_1, \ldots, A_k, \bar{y}_1, \ldots, \bar{y}_t, D)
\end{align*}\]

have to be collinear as well, i.e., satisfy $Q = P + \alpha \cdot (R - P)$.

To see this, recall that each condition $a_i \in A_i$ is a distribution over the values of the corresponding attribute $X_i$. Hence the above probabilities are actually expectations over these combinations of $X$-values. Formally, $P$ (and similarly $Q$ and $R$) is defined as

\[\begin{align*}
P &= \mathbb{E}_{X, X_1, \ldots, X_k}[\Pr(Y \mid X, X_1, \ldots, X_k, \bar{y}_1, \ldots, \bar{y}_t, D)] \\
Q &= \mathbb{E}_{X_1, \ldots, X_k}[\Pr(Y \mid X = x, X_1, \ldots, X_k, \bar{y}_1, \ldots, \bar{y}_t, D)] \\
R &= \mathbb{E}_{X, X_1, \ldots, X_k}[\Pr(Y \mid X = r, X_1, \ldots, X_k, \bar{y}_1, \ldots, \bar{y}_t, D)]
\end{align*}\]

Since expectations can be decomposed, we can equivalently write

\[\begin{align*}
P &= \mathbb{E}_X[\mathbb{E}_{X_1, \ldots, X_k}[\Pr(Y \mid X, X_1, \ldots, X_k, \bar{y}_1, \ldots, \bar{y}_t, D)]] \\
Q &= \mathbb{E}_X[\mathbb{E}_{X_1, \ldots, X_k}[\Pr(Y \mid X = x, X_1, \ldots, X_k, \bar{y}_1, \ldots, \bar{y}_t, D)]] \\
R &= \mathbb{E}_X[\mathbb{E}_{X_1, \ldots, X_k}[\Pr(Y \mid X = r, X_1, \ldots, X_k, \bar{y}_1, \ldots, \bar{y}_t, D)]]
\end{align*}\]

Based on the definition of the expectation, we then obtain $P = \sum_{x \in X} \Pr(x) \cdot g(x)$, where $g(x) = \mathbb{E}_{X_1, \ldots, X_k}[\Pr(Y \mid X = x, X_1, \ldots, X_k, \bar{y}_1, \ldots, \bar{y}_t, D)]$; similar for $Q$ and $R$. For our 3-valued example attribute $X$ we therefore have

\[\begin{align*}
P &= p_1 \cdot g(x_1) + p_2 \cdot g(x_2) + (1 - p_1 - p_2) \cdot g(x_3) \\[2.5]
Q &= q_1 \cdot g(x_1) + q_2 \cdot g(x_2) + (1 - q_1 - q_2) \cdot g(x_3) \\[2.6]
R &= r_1 \cdot g(x_1) + r_2 \cdot g(x_2) + (1 - r_1 - r_2) \cdot g(x_3) \\[2.7]
\end{align*}\]

Since $q = p + \alpha \cdot (r - p)$, we can derive from Equation 2.6

\[\begin{align*}
Q &= (p_1 + \alpha(r_1 - p_1))g(x_1) + (p_2 + \alpha(r_2 - p_2))g(x_2) \\
&\quad + (1 - (p_1 + \alpha(r_1 - p_1)) - (p_2 + \alpha(r_2 - p_2)))g(x_3) \\
&= (p_1 g(x_1) + p_2 g(x_2) + (1 - p_1 - p_2) g(x_3)) \\
&\quad + \alpha((r_1 g(x_1) + r_2 g(x_2) + (1 - r_1 - r_2) g(x_3)) \\
&\quad - (p_1 g(x_1) + p_2 g(x_2) + (1 - p_1 - p_2) g(x_3))).
\end{align*}\]
Together with Equations 2.5 and 2.7, we then obtain the desired result that
\[ Q = P + \alpha \cdot (R - P). \]

Intuitively, the lemma states the following: Assume entity \( y \) has a probability \( P \) based on condition \( p \) for attribute \( X \). Assume we also know that this probability will be \( R \) if the user changed the condition from \( p \) to a different distribution \( r \). Then each alternative condition \( q \) that is a linear combination of \( p \) and \( r \) will result in a probability \( Q \) that is proportionally between \( P \) and \( R \). Figure 2.16 illustrates this property for an example of five entities \( y_1 \) to \( y_5 \). For distribution \( p \) of attribute \( X \), \( y_1 \) has the highest probability and \( y_5 \) the lowest. For a different distribution \( r \) those probabilities are almost reversed. For any distribution \( q \) that is a linear combination of \( p \) and \( r \), each entity’s probability is proportionally between the corresponding probabilities for \( p \) and \( r \).

Lemma 2.3.2 implies strong limitations on how entities can change their relative rankings. As illustrated by entity pairs \( (y_1, y_2) \) and \( (y_4, y_5) \) in Figure 2.16, if some entity \( y \) is ranked below another entity \( y' \) in both \( L_p \) (ranking on the left) and \( L_r \) (ranking on the right), then the rank order will be same for any \( L_q \) “in-between”.
Lemma 2.3.3. Let \( p, r, \) and \( q = p + \alpha \cdot (r - p) \), \( 0 < \alpha < 1 \), be collinear conditions for attribute \( X \). And let the entities be ranked by effort-adjusted probability. If entity \( y \) is ranked below entity \( y' \) in both \( L_p \) and \( L_r \), then \( y \) is also ranked below \( y' \) in \( L_q \).

Proof. We use \( \phi \) and \( \phi' \) to denote the user effort for \( y \) and \( y' \), respectively (see Section 2.2). As before, let \( P, Q, \) and \( R \) denote the probability of entity \( y \) in \( L_p \), \( L_q \), and \( L_r \), respectively. We similarly define \( P', Q', \) and \( R' \) for \( y' \).

As shown above, the collinearity of \( p, q, \) and \( r \) implies collinearity of \( P, Q, \) and \( R \) (similarly for \( P', Q', \) and \( R' \)). More precisely, \( Q = P + \alpha \cdot (R - P) = (1 - \alpha)P + \alpha R \) and similarly \( Q' = (1 - \alpha)P' + \alpha R' \). This implies \( Q/\phi - Q'/\phi' = (1 - \alpha)(P/\phi - P'/\phi') + \alpha(R/\phi - R'/\phi') \). This difference is negative, because \( y' \) is ranked above \( y \) in both \( L_p \) and \( L_r \) and therefore \( P/\phi < P'/\phi' \) and \( R/\phi < R'/\phi' \). This in turn implies that \( y' \) is ranked above \( y \) in \( L_q \) as well.

2.3.6 Monotonicity of Ranking Distance

We can now prove the following result for sensitivity:

**Theorem 2.3.4.** Let \( p \) be the current query condition for attribute \( X \) and let \( q, r \in \mathcal{A} \) be two alternative conditions considered by the user, such that \( q = p + \alpha \cdot (r - p) \) for some \( 0 < \alpha < 1 \). \( L_p, L_q, \) and \( L_r \), respectively, denote the rankings obtained for these conditions, while keeping all other conditions unchanged. Then, if entities are ranked based on their probability or effort-adjusted probability (Section 2.2), it holds that \( \text{dst}(L_p, L_q) \leq \text{dst}(L_p, L_r) \). The same property also holds for distance measures \( \text{dst}_r \) and \( \text{dst}_{\text{AP}} \).

Proof. We first prove the theorem for Minkowski distance \( \text{dst} \). Consider Figure 2.16 for illustration. As \( \alpha \) is increased from 0 to 1, we obtain a series of different rankings between the entities due to their changing probabilities. In the example, for \( 0 < \alpha < 0.2 \) the ranking of probabilities for \( q \) is identical to the ranking for \( p \). At \( \alpha = 0.2 \), “adjacent” (in their ranking) entities \( y_2 \) and \( y_3 \) swap places. Then
at $\alpha = 0.3$, adjacent $y_2$ and $y_4$ swap places. Notice that sometimes more than two entities might swap places “at the same time” when multiple lines intersect, e.g., for $\alpha = 0.4$ in the example. The result of this many-entity swap can always be equivalently expressed as a series of binary swaps between adjacent entities (using the Bubble-sort algorithm). Consider again Figure 2.16. For $\alpha = 0.39$, the entity ranking is $(y_1, y_3, y_4, y_2, y_5)$; for $\alpha = 0.41$ it is $(y_4, y_3, y_1, y_5, y_2)$. The following sequence of binary swaps between adjacent entities transforms one ranking to the other: $y_1 \leftrightarrow y_3, y_1 \leftrightarrow y_4, y_2 \leftrightarrow y_5, y_3 \leftrightarrow y_4$.

Now consider the sequence of distinct rankings $L_0 (= L_p), L_1, L_2, \ldots, L_{k-1}, L_k (= L_r)$ defined by gradually increasing $\alpha$ from 0 to 1. Each ranking pair $(L_i, L_{i+1})$, $0 \leq i < k$, is identical except that two adjacent entities in $L_i$ are swapped in $L_{i+1}$. We show that this property implies that the Minkowski distance from $L_p$ to $L_i$ cannot be greater than the distance from $L_p$ to $L_{i+1}$.

Let $y$ and $y'$ be the adjacent entities that swapped ranks between $L_i$ and $L_{i+1}$. More precisely, both rankings are identical, except that $\rho_i(y) = \rho_i(y') - 1,$
CHAPTER 2. IMPRECISE INTERACTIVE QUERIES

\[ \rho_{i+1}(y) = \rho_{i+1}(y') + 1, \text{ and } \rho_{i+1}(y') = \rho_i(y). \] Now consider all possible cases for their ranking in \( L_p \) (see Figure 2.17):

**Case 1:** \( \rho_p(y) > \rho_i(y) \) and \( \rho_p(y') < \rho_i(y') \). This case is impossible, because it implies \( \rho_p(y) > \rho_p(y') \), which together with \( \rho_i(y) < \rho_i(y') \) and \( \rho_{i+1}(y) > \rho_{i+1}(y') \) violates Lemma 2.3.3.

**Case 2:** \( \rho_p(y) \leq \rho_i(y) \) and \( \rho_p(y') \geq \rho_i(y') \). Since all entities are at the same ranks in \( L_i \) and \( L_{i+1} \), except for \( y \) and \( y' \), the only difference between \( \text{dst}(L_p, L_i) \) and \( \text{dst}(L_p, L_{i+1}) \) are terms containing the ranks of \( y \) and \( y' \). These terms are \( |\rho_p(y) - \rho_i(y)|^d + |\rho_p(y') - \rho_i(y')|^d \) for \( \text{dst}(L_p, L_i) \) versus \( |\rho_p(y) - \rho_{i+1}(y)|^d + |\rho_p(y') - \rho_{i+1}(y')|^d \) for \( \text{dst}(L_p, L_{i+1}) \). Because of the case constraint and the fact that \( y \) and \( y' \) have swapped ranks between \( L_i \) and \( L_{i+1} \), it follows that \( |\rho_p(y) - \rho_{i+1}(y)|^d + |\rho_p(y') - \rho_{i+1}(y')|^d \) is equal to \( (|\rho_p(y) - \rho_i(y)| + 1)^d + (|\rho_p(y') - \rho_i(y')| + 1)^d \), which is greater than \( |\rho_p(y) - \rho_i(y)|^d + |\rho_p(y') - \rho_i(y')|^d \). Hence \( \text{dst}(L_p, L_i) < \text{dst}(L_p, L_{i+1}) \).

**Case 3:** \( \rho_p(y) \leq \rho_i(y) \) and \( \rho_p(y') < \rho_i(y') \). Observe that \( |\rho_p(y) - \rho_i(y)| \geq |\rho_p(y') - \rho_q(y')| \). This is due to the facts that \( y' \) cannot precede \( y \) in \( L_p \) (Lemma 2.3.3) and that both are ranked higher in \( L_p \) than in \( L_i \). Therefore, similar to the analysis in case 2, we obtain the following for the rank-difference terms that are different between \( \text{dst}(L_p, L_i) \) and \( \text{dst}(L_p, L_{i+1}) \):

\[
|\rho_p(y) - \rho_{i+1}(y)|^d + |\rho_p(y') - \rho_{i+1}(y')|^d \\
= (\rho_{i+1}(y) - \rho_p(y))^d + (\rho_{i+1}(y') - \rho_p(y'))^d \\
= (\rho_i(y) + 1 - \rho_p(y))^d + (\rho_i(y') - 1 - \rho_p(y'))^d \\
= (\rho_i(y) - \rho_p(y))^d + d(\rho_i(y) - \rho_p(y))^{d-1} + \cdots + 1 \\
+ (\rho_i(y') - \rho_p(y'))^d - d(\rho_i(y') - \rho_p(y'))^{d-1} + \cdots + (-1)^d \\
\geq (\rho_i(y) - \rho_p(y))^d + (\rho_i(y') - \rho_p(y'))^d \\
= |\rho_p(y) - \rho_i(y)|^d + |\rho_p(y') - \rho_i(y')|^d \\
\Rightarrow \text{dst}(L_p, L_i) \leq \text{dst}(L_p, L_{i+1})
\]

**Case 4:** \( \rho_p(y) > \rho_i(y) \) and \( \rho_p(y') \geq \rho_i(y') \). The analysis is symmetric to case
3.

These cases cover all possible rankings for \( L_p \). We can now inductively apply this argument, showing that \( \text{dst}(L_p, L_1) \leq \text{dst}(L_p, L_2) \leq \cdots \leq \text{dst}(L_p, L_k) = \text{dst}(L_p, L_r) \).

We now prove Theorem 2.3.4 for Kendall’s \( \tau \) distance and Average Precision distance.

**Proof.** Consider the same rankings \( L_i \) and \( L_{i+1} \) as used in the proof for the Minkowski distance. Since \( L_i \) and \( L_{i+1} \) are identical except for the adjacent entities \( y \) and \( y' \), which swap places, the number of inversions compared to \( L_0 \) is also identical, except for the pair \((y, y')\). Note that \( y \) is ranked above \( y' \) in \( L_i \), but below \( y' \) in \( L_{i+1} \). Lemma 2.3.3 implies that in \( L_0 \), \( y \) must have been ranked above \( y' \). Hence \( L_{i+1} \) has one additional inversion than \( L_i \) compared to \( L_0 \), completing the proof for \( \text{dst}_\tau(L_p, L_i) < \text{dst}_\tau(L_p, L_{i+1}) \).

For Average Precision distance, consider \( \text{dst}_{\text{AP}}(L_0, L_{i+1}) - \text{dst}_{\text{AP}}(L_0, L_i) = \frac{2}{n-1} \sum_{k=2}^{n} (C_{0,i}(k) - C_{0,i+1}(k)) / (k - 1) \). For simplicity, let \( h = \rho_i(y) = \rho_{i+1}(y') \) (and therefore \( h + 1 = \rho_i(y') = \rho_{i+1}(y) \)). Since \( L_i \) and \( L_{i+1} \) are identical except for the swap of adjacent entities \( y \) and \( y' \), \( C_{0,i}(k) = C_{0,i+1}(k) \) for all \( k \neq h, (h + 1) \). This implies \( \text{dst}_{\text{AP}}(L_0, L_{i+1}) - \text{dst}_{\text{AP}}(L_0, L_i) = \frac{2}{n-1} T \), where

\[
T = \frac{C_{0,i}(h)}{h - 1} + \frac{C_{0,i}(h + 1)}{h} - \frac{C_{0,i+1}(h)}{h - 1} - \frac{C_{0,i+1}(h + 1)}{h}.
\]

To see that \( T \) is non-negative, note that \( C_{0,i}(h) = C_{0,i+1}(h + 1) \), because the entities ranked above \( y \) are the same in both \( L_i \) and \( L_{i+1} \), except for \( y' \), which ranks above \( y \) only in \( L_{i+1} \). However, we already showed that in \( L_0 \), \( y \) has to be ranked above \( y' \), therefore \( y' \) does not increase \( C_{0,i+1}(h + 1) \) compared to \( C_{0,i}(h) \). A similar analysis shows that \( C_{0,i}(h + 1) = C_{0,i+1}(h) + 1 \), because in \( L_{i+1} \), \( y' \) is not ranked above \( y \) any more, reducing the number of agreements with the ranking in \( L_0 \) by one.

Using these results, and the fact that by definition \( C_{0,i+1}(h + 1) \leq h \) (there are only \( h \) entities total above rank \( h + 1 \)), it is easy to show that \( T \geq 0 \), and therefore \( \text{dst}_{\text{AP}}(L_0, L_{i+1}) - \text{dst}_{\text{AP}}(L_0, L_i) \geq 0 \).
2.4 Recommending Additional Conditions

It is inherently difficult to determine how much any of the remaining attributes, i.e., those for which the user has not specified a condition yet, would improve result quality (left table in Figure 1.3). We call this component the attribute ranker or \texttt{aRank}. The usefulness of an attribute \(X\) depends on the condition \(a \in A\) the user will enter for it—which is unknown. We address this problem by modeling the unknown future condition as a random variable. Merlin estimates the probability that the user will enter distribution \(a \in A\) based on the information provided so far as

\[
\Pr(A = a \mid A_1, \ldots, A_k, \bar{y}_1, \ldots, \bar{y}_l, D).
\]

Merlin then estimates the entity probabilities given the predicted distribution \(a\):

\[
\Pr(Y \mid A = a, A_1, \ldots, A_k, \bar{y}_1, \ldots, \bar{y}_l, D).
\]

This leaves another challenge: Even knowing the likely new entity probabilities is not helpful unless one can determine which of them are the best. Doing so would be trivial if Merlin knew which entities the user is looking for—pick the ones where the entities of interest have the highest probabilities. Without that information, we can only rely on general quality measures that evaluate how well “winning” entities are separated from “losing” ones. More precisely, it is desirable to have a result where the probability is high for a few entities and near-zero for all others: likely answers and unlikely answers are well-separated and the few top-ranked entities shown to the user have a comparably high aggregate probability mass.

\textit{Entropy} directly captures this intuition. Using \(p_y\) as shorthand for \(\Pr(Y = y \mid A, A_1, \ldots, A_k, \bar{y}_1, \ldots, \bar{y}_l, D)\), entropy for the set of entities is defined as

\[
-\sum_{y \in Y} p_y \cdot \log_2(p_y).
\]

In general, entropy is low if there are few high-probability entities and many low-probability ones. It is high if many entities have similar probability. (For this reason entropy is widely used for selection of split attributes in decision trees [25].) The expected improvement in the quality of the entity ranking is then measured as the expected entropy reduction for attribute \(X\). Instead of
entropy, one could also use other common measures of “purity” for classification problems, including Gini [25], or measures based on expected user effort [41].

2.5 Probability Estimation in Realtime

Given user-provided conditions $A_1, A_2, \ldots, A_k$ (which are distributions over the possible values of the corresponding attributes $X_1, \ldots, X_k$) and explicitly rejected entities $y_1, \ldots, y_l$, Merlin needs to estimate the probabilities of entities, $\Pr(Y \mid A_1, \ldots, A_k, \bar{y}_1, \ldots, \bar{y}_l, D)$, and of new conditions the user might specify, $\Pr(A \mid A_1, \ldots, A_k, \bar{y}_1, \ldots, \bar{y}_l, D)$. Formulas of the type $\Pr(C \mid X_1, X_2, \ldots, X_k, D)$ define the posterior probability of a class $C$ in Bayesian classification [35]. In addition to Bayesian classification techniques, it has been shown that virtually all popular classification methods such as SVMs, artificial neural networks, and decision tree ensembles can be modified to output such probabilities [39]. Based on this observation, we can in principle leverage almost any classification technique using the following approach for estimating $\Pr(Y \mid A_1, \ldots, A_k, \bar{y}_1, \ldots, \bar{y}_l, D)$:

- Using data set $D$, train a classification model $M(X_1, X_2, \ldots, X_k)$ that predicts the probability of each entity for a given input vector $(x_1, x_2, \ldots, x_k) \in X_1 \times X_2 \times \cdots \times X_k$. (If $D$ contains probabilistic data, then Merlin can use classification techniques for uncertain data [48]. Alternatively, one can transform a probabilistic data set $D$ to a data set without uncertainty by sampling multiple training records from each uncertain data record.)

- At query time, given user-provided conditions $a_1 \in A_1, \ldots, a_k \in A_k$, sample value $x_i$ from distribution $a_i$ for $i = 1, \ldots, k$. Then compute $M(x_1, x_2, \ldots, x_k)$, i.e., the probability for each entity, by running $(x_1, x_2, \ldots, x_k)$ through model $M$. Scale the probabilities of all entities $y \in Y - \{y_1, \ldots, y_l\}$ proportionally so that they sum up to 1.

Computing the entity probabilities is essential to the entity ranker. The pseudocode of the entity ranker is presented in Algorithm 1. This algorithm shows how
Algorithm 1: Entity Ranker Algorithm

1: procedure E\textsc{Rank}(M(X_1, \ldots, X_k), a_1, \ldots, a_k, \bar{y}_1, \ldots, \bar{y}_l)
2:  for $i = 1 \rightarrow s_1$ do
3:  Sample point $\alpha_i = (x_1, \ldots, x_k)$ such that each $x_\lambda$ is drawn from the corresponding distribution $a_\lambda$
4:  Run $\alpha_i = (x_1, \ldots, x_k)$ through model $M(X_1, \ldots, X_k)$ to find $Pr(Y|X_1 = x_1, \ldots, X_k = x_k, \bar{y}_1, \ldots, \bar{y}_l, D)$
5:  end for
6:  Compute $Pr(Y|A_1 = a_1, \ldots, A_k = a_k, \bar{y}_1, \ldots, \bar{y}_l, D)$ as the average of the probabilities found for the $s_1$ sampled points
7:  Rank entities based on $Pr(Y|A_1 = a_1, \ldots, A_k = a_k, \bar{y}_1, \ldots, \bar{y}_l, D)$
8: end procedure

to compute entity probabilities using the above strategy. When the user-provided conditions are distributions over the values in the domain of the attributes, $s_1$ points are sampled from these distributions and the entity probabilities are found using model $M(X_1, \ldots, X_k)$. Then, the desired probabilities are obtained by computing the expectation over the entity probabilities at the $s_1$ sampled points. Finally, the entities are ranked based on their probabilities.

We can similarly train and use a model $M_X(X_1, X_2, \ldots, X_k)$ to predict the user’s input for an attribute $X$ for which she has not yet provided a query condition. Given input $(x_1, x_2, \ldots, x_k)$ (i.e., values of some of the other attributes), it returns $Pr(A|A_1, \ldots, A_k, \bar{y}_1, \ldots, \bar{y}_l, D)$, i.e., a probability distribution over possible user inputs for attribute $X$. These probabilities are used in the attribute ranker. The pseudocode of the attribute ranker, shown in Algorithm 2, is more complicated than the pseudocode for entity ranker. For each unspecified attribute, first the attribute model is used to predict attribute probabilities as an expectation over $s_2$ sampled points. Then, for each sample, the entity model returns expectations over $s_1 \cdot s_2$ sampled points. In the final step, attributes are ranked based on a function of their probabilities, i.e. information gain.
Algorithm 2: Attribute Ranker Algorithm

1: procedure A\textsc{Rank}(M_X(X_1, ..., X_k), M'(X, X_1, ..., X_k), a_1, ..., a_k, \bar{y}_1, ..., \bar{y}_l)
2: for all unspecified attribute X do
3: for i = 1 \rightarrow s_1 do
4: Sample point \alpha_i = (x_1, ..., x_k) such that each \( x_\lambda \) is drawn from the corresponding distribution \( a_\lambda \)
5: Run \( \alpha_i = (x_1, ..., x_k) \) through model \( M_X(X_1, ..., X_k) \) to find \( Pr(A = a | X_1 = x_1, ..., X_k = x_k, \bar{y}_1, ..., \bar{y}_l, D) \)
6: for j = 1 \rightarrow s_2 do
7: Sample point \( \beta_j = (x, x_1, ..., x_k) \) such that each \( x_\lambda \) is drawn from the corresponding distribution \( a_\lambda \) and \( x \) is drawn from its corresponding distribution \( a \)
8: Run \( \beta_j = (x, x_1, ..., x_k) \) through model \( M'(X, X_1, ..., X_k) \) to find \( Pr(Y | X = x, X_1 = x_1, ..., X_k = x_k, \bar{y}_1, ..., \bar{y}_l, D) \)
9: end for
10: end for
11: Compute \( Pr(Y | A = a, A_1 = a_1, ..., A_k = a_k, \bar{y}_1, ..., \bar{y}_l, D) \) as the average of the probabilities found for the \( s_1 \cdot s_2 \) sampled points
12: Compute information gain for attribute X
13: end for
14: Rank attributes based on their information gain
15: end procedure

2.5.1 Challenges

While conceptually straightforward, the problem lies in guaranteeing interactive response time. Recall that user-specified conditions are distributions over the values of the corresponding attribute domain. Hence the probabilities of interest are actually expectations, which in practice are computed by sampling from the respective domains. Consider computing the entity probabilities \( Pr(Y | A_1, ..., A_k, \bar{y}_1, ..., \bar{y}_l, D) \) at query time as described in Algorithm 1. If \( s_1 \) inputs \((x_1, x_2, ..., x_k)\) are sampled to compute the average probability of each entity, then the total cost of the entity ranker is

\[
T_{\text{eRank}} = s_1 \cdot c_M + u
\]  

(2.8)

where \( c_M \) is the time it takes model \( M \) to make a single prediction and \( u \) is the time needed for ranking the entities, which is independent of the model size.
CHAPTER 2. IMPRECISE INTERACTIVE QUERIES

This formula is obtained from Algorithm 1 as follows. After the user enters new information, Merlin first has to access the entity prediction model to compute the probabilities of all entities (time: \( s_1 \cdot c_M \)). Then it sorts the entities (time: \( u \)).

Similarly, the time for attribute ranking or recommending additional conditions (Section 2.4), is obtained from Algorithm 2. In this algorithm, entity probabilities have to be computed many more times: For each of the \( (m - k) \) attributes \( X \) without user-specified condition, model \( M_X \) returns a distribution over possible user conditions \( a \in A \); and for a random sample of \( s_2 \) conditions from this output of \( M_X \), Merlin has to compute \( \Pr(\bar{Y} | A, A_1, \ldots, A_k, \bar{y}_1, \ldots, \bar{y}_l, D) \) using a model \( M'(X, X_1, X_2, \ldots, X_k) \) similar to \( M(X_1, X_2, \ldots, X_k) \). Hence the total prediction cost becomes

\[
T_{a\text{Rank}} = (m - k) \cdot s_1 (c_{M_X} + s_2 \cdot c_{M'}) + g + u' \tag{2.9}
\]

where \( g \) is the cost for computing information gain of the unspecified attributes and \( u' \) is the time needed for sorting the attributes based on their information gain. Both \( g \) and \( u' \) are independent of the model size. In this formula, we have assumed for the sake of simplicity that the model cost \( c_{M_x} \) is the same for all attributes \( X \).

While \( s_1 \) and \( s_2 \) are easily tuneable, \( c_{M'} \) and \( c_{M_x} \) are usually fixed or even hard to predict, depending on the model type used. We discuss in Section 2.5.3 how to make them tuneable.

In addition to prediction cost, one also has to consider model training time and storage cost. It often is not feasible to pre-compute and store models like \( M \) and \( M_X \) for all possible subsets of the set of attributes. On the other hand, training models on-the-fly can significantly increase system response time. (Note that for large data sets, state-of-the-art data mining models usually cannot be trained in a few seconds.) We discuss our solution next.

2.5.2 Solution: Bagged Tree Ensembles

We propose to use bagged decision tree ensembles [9] for probability estimation. A decision tree recursively partitions the data space, attempting to find partitions with
high purity, i.e., where one class clearly dominates over all others. Each non-leaf node in the tree splits the data space on some attribute. Tree traversal for making a prediction starts at the root and proceeds like in a standard search tree. The leaf nodes contain predictions based on the distribution of the data records that fall into the corresponding region of the data space. Details can be found in any data mining textbook [25]. A bagged tree ensemble consists of many such trees, each trained on an independent bootstrap sample of the training data. To make a prediction for a given input, all trees are traversed and their individual outputs are averaged. We chose bagged trees for several reasons.

1. Trees can handle any attribute type. Bagged trees are robust against noise and overfitting and have been shown to return excellent probabilities “out-of-the-box” [39]. Trees also can naturally deal with missing values, therefore one can use a model trained for input \((X_1, \ldots, X_m)\) to make predictions for any subset of these attributes. This eliminates the problem of on-the-fly model training or pre-computing a large number of models for different attribute subsets.

2. Due to their structure of splitting on an attribute at a time, trees can compute expectations like \(\Pr(Y \mid A_1, \ldots, A_k, \bar{y}_1, \ldots, \bar{y}_l, D)\) in a single pass. At a split node whose attribute value is 100% certain, i.e., the user specified a condition where a single value in the domain has probability 1.0, the entire “weight” follows the corresponding branch. For imprecise conditions, the “weight” is partitioned according to user-specified probability distribution and each partial weight is sent down the corresponding branch. If the user has not specified a condition for the split attribute, partial weights are determined by the training data distribution of the attribute in the corresponding region of the data space (which is stored in the node). It is easy to show that this is consistent with the desired computation of the expected entity probabilities when taking the expectation over all uncertain input values. Hence, there is no need to sample from \(A_1, \ldots, A_k\), i.e., \(s_1\) in Equations 2.8 and 2.9 is
2.5.3 **Controlling Tree Response Time**

A major advantage of tree-based methods over other popular classification techniques is that their response time is reasonably predictable and tuneable, because it is directly determined by the number of nodes accessed. This is crucial for guaranteeing interactive response time. Let *Full Tree* refer to a bagged ensemble consisting of $T$ individual trees, each with at most $L$ levels. By using only $t \leq T$ of these trees and limiting access to the top $l \leq L$ levels, we can reduce cost approximately proportionally with the total number of nodes accessed. To be able to stop at a non-leaf level, Merlin also stores the corresponding class distribution in each inner node of the trees. Our goal is to set $(t, l)$ so that Merlin responds within a user-specified time. Notice that as the user enters query conditions, $t$ and $l$ might need to be changed adaptively. For instance, a condition assigning zero probability to some values of an attribute’s domain effectively prunes the corresponding subtree(s), reducing the number of nodes accessed. Figure 2.19 illustrates tree pruning. The tree at the top is the initial tree where the user has not yet specified the value of attribute billLength. When the user assigns probability distribution $(0, 0.6, 0.4)$ to this attribute, the branch where billLength is equal to 1 is pruned and weights...
2.5. **PROBABILITY ESTIMATION IN REALTIME**

Figure 2.19: Tree pruning due to assigning zero probability to some values of an attribute equal to 0.6 and 0.4 are assigned to the branches having value 2 and 3 respectively. When tree pruning happens, response time drops and the initial limits for $t$ and $l$ can be increased. Since our proposed tree ensemble is capable of adapting to the time threshold, we call it *Adaptive Tree*.

To determine if it is safe to increase $l$ or $t$, Merlin needs to estimate how many more tree nodes will be accessed if $t$ increases to $t'$ and $l$ to $l'$. The former is fairly simple: since all trees are trained on bootstrap samples of the same size and similar data distribution, we can estimate the number of nodes accessed in a newly added limited tree quite accurately as the average of the already used $t$ limited trees. The effect of a level increase is more difficult to estimate, because the sub-trees in the newly added levels are typically not balanced. However, since we already accessed
all nodes up to level \( l \), we know exactly how many children will be accessed at
level \( l + 1 \). Hence as long as Merlin increases \( l \) by at most 1 level at a time, it
can accurately predict the number of nodes accessed after the level-limit increase.
Figure 2.20 illustrates this idea. The white nodes are those that are already accessed
in the Adaptive Tree. The light grey nodes are not yet accessed, but the tree keeps
track of their count and accesses them when increasing the number of levels from \( l \)
to \( l + 1 \). The algorithm is totally unaware of the number of nodes in the next levels,
i.e. the dark grey nodes.

The only difficult case occurs when the user modifies or completely removes a
previously specified condition, say on attribute \( X \). After this modification, a child \( c \)
of a node splitting on \( X \) that had zero probability mass before might now have non-
zero probability and hence would be accessed. Since Merlin might not know how
many nodes will be accessed in the corresponding sub-tree, it takes a conservative
approach of adding one new level of the sub-tree at a time. It does so by adding a
“local” level limit \( l_c \) for \( c \), setting it initially to the level of \( c \). This way in the next
round, i.e., after the next user interaction, Merlin only accesses root node \( c \) of this
sub-tree. After accessing \( c \), Merlin knows the number of its children and hence can
increase the local level limit to \( l_c + 1 \). This process continues until the local limit
reaches global limit \( l \), at which point it is discarded. Figure 2.21 illustrates this
idea for the pruned tree from Figure 2.19. Assume the user modifies the current
condition of $(0, 0.6, 0.4)$ to $(0.1, 0.5, 0.4)$ which no longer assigns zero probability to billLength=1. Since the algorithm does not know how many nodes would be accessed in this subtree, it uses $l_c = 2$ and increases its value in 2 rounds until it reaches the desired global level limit $l = 4$.

Knowing the number of nodes accessed, Merlin can estimate system response time. The three major computations performed by Merlin in response to user input are (see also Figure 2.2) ranking of entities, ranking of attributes, and computing sensitivity. Equations 2.8 and 2.9 are the generic formulas for computing the time for entity ranking and attribute ranking. We now derive the concrete formulas for bagged tree ensembles, where model access time is determined by the number of tree nodes accessed. Let $n_y$ denote the number of nodes accessed in the Adaptive Tree used to predict the entity probabilities, and let $\theta$ denote the average time for accessing a single tree node. Then the time for entity ranking becomes

$$T_{e\text{Rank}} = n_y\theta + u.$$

Let $n_a$ denote the total number of nodes accessed in the Adaptive Trees for predicting future input for unspecified attributes. Then the time for attribute ranking becomes

$$T_{a\text{Rank}} = n_a\theta + (m - k)s_2n_y\theta + g + u'.$$

This formula is obtained as follows. First, all models for unspecified attributes are accessed to obtain probability distributions for possible conditions the user might input for them (time: $n_a\theta$). For each of these $m - k$ attributes, $s_2$ conditions are sampled (see Section 2.5.1), and for each of them the corresponding new entity probabilities are computed (time: $(m - k)s_2n_y\theta$). Note that $g$ is negligible compared to computation of the entity probabilities it needs as input. The cost for sorting of the $m - k$ unspecified attributes, $u'$, is also negligible compared to predicting the distribution and resulting entity probabilities for each of these attributes. In our current Merlin implementation, entity and attribute ranker are both executed after each user input. Hence, a user-specified response-time threshold is applied to
Figure 2.21: When user modifies the condition of billLength, the previously pruned branch is accessed again, but initially with a local level limit of $l_c$. In the next 2 rounds $l_c$ is gradually increased until it reaches the global level limit $l$. 
the total computation time

\[ T = T_{e\text{Rank}} + T_{a\text{Rank}} + z, \] (2.10)

where \( z \) accounts for the constant, i.e., independent of model size, overhead for all other tasks related to reading the user’s response and updating the UI.

Sensitivity analysis is only executed on demand, and it has its own user-specified response time threshold. The time for sensitivity analysis is

\[ T_{\text{sens}} = k s_3 (v + n_y \theta + u + w) + z. \]

It is obtained as follows. For a specified attribute, an alternative condition is sampled from \( \mathcal{A} \) (time: \( v \)), followed by the computation of the resulting entity probabilities (time: \( n_y \theta \)). This is followed by sorting of the entities (time: \( u \)) and computing the distance to the original ranking (time: \( w \)). This computation is performed for each of the specified \( k \) attributes and \( s_3 \) samples of alternative conditions for each of them. Both \( v \) and \( w \), like \( u \), are constant as they do not depend on the model.

All variables in the above formulas can be measured as Merlin interacts with the user. To estimate the response time for a different model parameter combination \((l', t')\), Merlin simply plugs in the corresponding new node counts \( n_y \) and \( n_a \), estimated as discussed above. It can choose any combination \((l', t')\) for which the estimated computation time is below the corresponding response-time limit. Our experiments show that we indeed can successfully guarantee interactive response times.

### 2.5.3.1 Tuning Number of Trees and Levels

When deciding about \( l \) and \( t \), the smaller each of these values, the lower computation time. However, reducing computation time comes at a cost. Even if the full bagged tree ensemble returns good probabilities, reducing tree size and number of trees can reduce prediction quality.

For a given response-time limit, there are often many \((l, t)\) combinations that satisfy it. Among them, we only need to consider the “skyline”. More precisely,
if two such combinations satisfy the response-time constraint, but in the first each parameter value is greater or equal to the corresponding value in the second, then the second does not need to be considered. To select between the incomparable configurations on the skyline, we explore four strategies.

**Prefer increasing \( t \) over increasing \( l \):** At each step, increase the number of trees in the ensemble as much as the realtime threshold allows, until \( t \) reaches its maximum value \( T \). Only then start considering an increase of the number of levels. This strategy is based on the fact that adding more trees to the ensemble tends to improve the predictions by reducing the variance without affecting the bias of the model.

**Prefer increasing \( l \) over increasing \( t \):** At each step, add another level if the time threshold allows, until all levels are used in the trees. Only then consider increasing the number of trees. This strategy is motivated by the observation that adding more levels tends to reduce the bias of the model, but it might experience overall accuracy loss due to overfitting. To overcome the potential overfitting problem, we propose the third strategy.

**Increase \( t \) only if increasing \( l \) is not possible:** At each step, add another level if the time threshold allows, otherwise increase the number of trees as much as possible. Therefore, while individual trees grow to reduce the bias, more trees are added to the ensemble to reduce the variance. In this strategy, the relative growth of \( l \) and \( t \) highly depends on the data and in the extreme cases, it can be equal to the first or the second strategies. The former occurs if adding a level always exceeds the time limit in the beginning and therefore, more and more trees are added to the ensemble until there is no more left. Then gradually, increasing \( l \) becomes possible. The latter happens if a level can always be added within the time limit. Hence, \( t \) only increases after all individual trees are full trees. The first scenario usually happens when many attributes are either unknown or uncertain and therefore not many nodes are pruned. In this case, adding a level significantly increases the number of current nodes, while adding more trees potentially adds fewer nodes. On the other hand, the second scenario happens if most attribute values are certain
and therefore, most branches are pruned.

**Constant Ratio:** At each step, increase either $l$ or $t$ such that ratio $t/l$ is as close to a constant $c$ as possible. Compared to the other three strategies, this strategy guarantees to balance the increase in the limit of the number of levels and the number of trees. Therefore, it looks after both bias and variance in a more balanced way. By adjusting the value of $c$, one can emphasize more one or the other.

The appropriate choice of $c$ depends on the problem and experimental results. In pre-processing, the system admin can measure accuracy of the ensemble for different ratios, each time trying several $(l, t)$ pairs for a given ratio. The best one is then chosen to be used by Merlin at runtime.

In this strategy, we first estimate the number of trees $t'$ assuming that the number of levels is constant. This gives us a ratio $c' = t'/l$. Then we increment the number of levels by 1 and compute the number of trees $t''$. Note that $t''$ might be smaller than $t$, the current number of trees. This would give another ratio $c'' = t''/\left(l + 1\right)$. We compare $c'$ and $c''$ to the constant $c$. The closest value determines the winner. For example, assume that the system admin has decided to set the constant ratio $c$ to 3. Also suppose that the largest adaptive tree that guarantees the interactive response time and its ratio $t/l$ is close to $c = 3$ has limits $t = 12$ and $l = 4$ with approximately 80 nodes in each tree. Assume that specifying the value of the first attribute prunes almost $1/4$ of the tree nodes, i.e. each tree has about 60 nodes now. Therefore, by keeping the number of levels at 4, we can increase the number of trees to 16, which results in $c' = 16/4 = 4$. Assuming that increasing number of levels to 5, almost doubles the number of nodes in each tree, i.e. 120 nodes, we need to reduce the number of trees to 8, which in turn results in $c'' = 8/5 = 1.6$. In the final step $c' = 4$ and $c'' = 1.6$ are compared to $c = 3$ and since $|4 - 3| < |1.6 - 3|$, the strategy chooses to increase the number of trees to 16 and keep the level limit at 4.


2.6 Experiments

The goal of the experiments is to provide a proof of concept for three important properties of Merlin: (1) improved efficiency of sensitivity analysis due to Theorem 2.3.4, (2) quality of the recommendations for additional conditions, and (3) guarantee of interactive system response times. In all experiments, the ranker for $Y$ uses entity probabilities and the ranker for $A$ uses entropy (see Figure 2.2). All algorithms were coded using Java. Our experiments were conducted on a low-end dual-core 2.13GHz Intel PC running Linux with 4GB of RAM and a 500GB IDE HDD.

2.6.1 Data and Models

Merlin is evaluated on a data set provided by the Cornell Lab of Ornithology. It reports sightings of 367 different bird species in North America, each described by 164 categorical attributes encoding multi-valued attributes such as beak length, bird size and shape, and Boolean attributes describing color and color pattern for different body parts. Tables 2.2 and 2.3 show a list of these attributes along with their domains.

Each record reports an individual bird observation generated as follows. First, an observation record is randomly selected from the real eBird data set [17], containing actual observations reported by citizen scientists. While eBird lists the species of the observed bird, it does not contain individual bird properties such as color and size. Individual bird features were added to the eBird record by sampling from a feature distribution that was carefully defined by the domain experts for every single species. Notice that we removed location and time information from the records. (Based on location and time of observation, many bird species can be eliminated from consideration. Merlin discovers this automatically, recommending spatio-temporal attributes to the user first. While this is desired in practice and leads to quicker identification, we wanted to make the problem more challenging to show that Merlin works well even when the most important attributes for
### 2.6. EXPERIMENTS

Table 2.2: List of attributes and their domains in the data set provided by Cornell Lab of Ornithology - Part 1

<table>
<thead>
<tr>
<th>Name</th>
<th>Domain</th>
<th>Name</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>{1, 2, 3, 4, 5, 6, 7, 8, 9}</td>
<td>shape</td>
<td>{waterbirds/ducks/herons, songbirds, hawks/owls, grouse/quail}</td>
</tr>
<tr>
<td>beak_length</td>
<td>{shorter, equal, longer}</td>
<td>main_gray</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>main_black</td>
<td>{Y, N}</td>
<td>main_brown</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>main_white</td>
<td>{Y, N}</td>
<td>main_red</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>main_buff</td>
<td>{Y, N}</td>
<td>main_yellow</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>main_rufous</td>
<td>{Y, N}</td>
<td>main_olive</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>main_orange</td>
<td>{Y, N}</td>
<td>main_purple</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>main_green</td>
<td>{Y, N}</td>
<td>main_iridescent</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>main_blue</td>
<td>{Y, N}</td>
<td>breast_black</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>main_pink</td>
<td>{Y, N}</td>
<td>breast_white</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>breast_gray</td>
<td>{Y, N}</td>
<td>breast_brown</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>breast_brown</td>
<td>{Y, N}</td>
<td>breast_rufous</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>breast_red</td>
<td>{Y, N}</td>
<td>breast_orange</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>breast_yellow</td>
<td>{Y, N}</td>
<td>breast_green</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>breast_olive</td>
<td>{Y, N}</td>
<td>breast_blue</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>breast_purple</td>
<td>{Y, N}</td>
<td>breast_pink</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>breast_iridescent</td>
<td>{Y, N}</td>
<td>breast_solid</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>breast_streaked</td>
<td>{Y, N}</td>
<td>breast_spotted</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>breast_barred</td>
<td>{Y, N}</td>
<td>belly_black</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>belly_white</td>
<td>{Y, N}</td>
<td>belly_gray</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>belly_buff</td>
<td>{Y, N}</td>
<td>belly_brown</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>belly_rufous</td>
<td>{Y, N}</td>
<td>belly_red</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>belly_orange</td>
<td>{Y, N}</td>
<td>belly_yellow</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>belly_green</td>
<td>{Y, N}</td>
<td>belly_olive</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>belly_blue</td>
<td>{Y, N}</td>
<td>belly_purple</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>belly_pink</td>
<td>{Y, N}</td>
<td>belly_iridescent</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>belly_solid</td>
<td>{Y, N}</td>
<td>belly_streaked</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>belly_spotted</td>
<td>{Y, N}</td>
<td>belly_barred</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>back_black</td>
<td>{Y, N}</td>
<td>back_white</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>back_gray</td>
<td>{Y, N}</td>
<td>back_buff</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>back_brown</td>
<td>{Y, N}</td>
<td>back_rufous</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>back_red</td>
<td>{Y, N}</td>
<td>back_orange</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>back_yellow</td>
<td>{Y, N}</td>
<td>back_green</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>back_olive</td>
<td>{Y, N}</td>
<td>back_blue</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>back_purple</td>
<td>{Y, N}</td>
<td>back_pink</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>back_iridescent</td>
<td>{Y, N}</td>
<td>back_solid</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>back_streaked</td>
<td>{Y, N}</td>
<td>back_spotted</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>back_barred</td>
<td>{Y, N}</td>
<td>wing_black</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>wing_white</td>
<td>{Y, N}</td>
<td>wing_gray</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>wing_buff</td>
<td>{Y, N}</td>
<td>wing_brown</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>wing_rufous</td>
<td>{Y, N}</td>
<td>wing_red</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>wing_orange</td>
<td>{Y, N}</td>
<td>wing_yellow</td>
<td>{Y, N}</td>
</tr>
</tbody>
</table>
Table 2.3: List of attributes and their domains in the data set provided by Cornell Lab of Ornithology - Part 2

<table>
<thead>
<tr>
<th>Name</th>
<th>Domain</th>
<th>Name</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>wing_green</td>
<td>{Y, N}</td>
<td>wing_olive</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>wing_blue</td>
<td>{Y, N}</td>
<td>wing_purple</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>wing_pink</td>
<td>{Y, N}</td>
<td>wing_iridescent</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>head_black</td>
<td>{Y, N}</td>
<td>head_white</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>head_gray</td>
<td>{Y, N}</td>
<td>head_buff</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>head_brown</td>
<td>{Y, N}</td>
<td>head_rufous</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>head_red</td>
<td>{Y, N}</td>
<td>head_orange</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>head_yellow</td>
<td>{Y, N}</td>
<td>head_green</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>head_olive</td>
<td>{Y, N}</td>
<td>head_blue</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>head_purple</td>
<td>{Y, N}</td>
<td>head_pink</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>head_iridescent</td>
<td>{Y, N}</td>
<td>bill_black</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>bill_white</td>
<td>{Y, N}</td>
<td>bill_gray</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>bill_buff</td>
<td>{Y, N}</td>
<td>bill_rufous</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>bill_orange</td>
<td>{Y, N}</td>
<td>bill_orange</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>bill_green</td>
<td>{Y, N}</td>
<td>bill_blue</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>bill_pink</td>
<td>{Y, N}</td>
<td>bill_pink</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>tail_black</td>
<td>{Y, N}</td>
<td>tail_white</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>tail_gray</td>
<td>{Y, N}</td>
<td>tail_buff</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>tail_brown</td>
<td>{Y, N}</td>
<td>tail_rufous</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>tail_red</td>
<td>{Y, N}</td>
<td>tail_orange</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>tail_yellow</td>
<td>{Y, N}</td>
<td>tail_green</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>tail_olive</td>
<td>{Y, N}</td>
<td>tail_blue</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>tail_purple</td>
<td>{Y, N}</td>
<td>tail_pink</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>tail_iridescent</td>
<td>{Y, N}</td>
<td>leg_black</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>leg_white</td>
<td>{Y, N}</td>
<td>leg_gray</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>leg_buff</td>
<td>{Y, N}</td>
<td>leg_rufous</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>leg_orange</td>
<td>{Y, N}</td>
<td>leg_orange</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>leg_green</td>
<td>{Y, N}</td>
<td>leg_green</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>leg_blue</td>
<td>{Y, N}</td>
<td>leg_blue</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>leg_pink</td>
<td>{Y, N}</td>
<td>leg_iridescent</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>bird_feeder</td>
<td>{Y, N}</td>
<td>cavity/nestbox</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>swimming_wading</td>
<td>{Y, N}</td>
<td>on_ground</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>in_air</td>
<td>{Y, N}</td>
<td>perched_in_open</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>in_thick_vegetation</td>
<td>{Y, N}</td>
<td>in_flock</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>vocalizing</td>
<td>{Y, N}</td>
<td>diving</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>soaring</td>
<td>{Y, N}</td>
<td>foraging_in_bush</td>
<td>{Y, N}</td>
</tr>
<tr>
<td>flycatching/hawking</td>
<td>{Y, N}</td>
<td>hovering</td>
<td>{Y, N}</td>
</tr>
</tbody>
</table>
2.6. EXPERIMENTS

bird identification are removed.) We sampled 3 million records, using 2 million of
them for model training and the other 1 million for testing.

The models used in all experiments are bagged decision tree ensembles consisting
of 100 individual trees, each grown so that nodes with fewer than 500 records
are not split any further (i.e., they are leaves). Full Tree and Adaptive Tree refer to
the complete bagged model and the proposed model with adaptive response time,
respectively. Since there is a tradeoff between response time threshold and result
quality for the Adaptive Tree, we study two cases: one with a generous 5 sec for
the system to respond (A5) and the other with a tighter 1 sec system response time
limit (A1). Because of these thresholds, A5 starts out with 18 (out of 100) individ-
ual trees, each restricted to 5 levels. A1 starts out with a smaller model, consisting
of 12 trees and 3 levels. The strategy to grow the Adaptive Tree in all experiments
is to follow a Constant Ratio with $c = T/L = 3.5$.

In all experiments, user input is generated by selecting a record from the test
data. (Recall that these records are generated from actual user-reported observa-
tions.) For a precise condition, the user provides the corresponding value from the
test record with probability mass 1. For an imprecise condition, only 75% of the
probability mass goes to the true value and the rest is divided between other values
in the attribute’s domain. The Imprecision parameter controls how many attributes
have an imprecise condition. Setting Imprecision=$p$ implies that with probability
$p$ the user provides an imprecise condition for an attribute. This decision is made
independently for each attribute.

2.6.2 Sensitivity Analysis

We compare our algorithm that exploits Theorem 2.3.4 to the naive algorithm. The
set of alternative conditions considered, $\mathcal{A}$, is set to the entire space of possible
probability distributions for the attribute examined. Conditions on all attributes are
imprecise, except the original condition for the attribute of interest is the uniform
distribution to model a user completely unsure about the true value. We sample dis-
tributions randomly from $\mathcal{A}$. The naive algorithm uses the sampled values, while our proposed algorithm exploits Theorem 2.3.4 and instead uses the corresponding point on the edge, obtained where the ray connecting the original point to the sampled point intersects with the edge of $\mathcal{A}$. 

Figure 2.22: Naive vs. efficient sensitivity estimation for a 9-valued attribute

Figure 2.23: Naive vs. efficient sensitivity estimation for a Boolean attribute
For computing the distance between rankings, we used Minkowski distance with $d = 2$. Figure 2.22 shows the maximum distance between the original ranking and the rankings obtained based on the sampled conditions for bird size, which has 9 values. After sampling only 4 points, the efficient algorithm discovered a distance within 5% of the final maximum distance it finds, while the naive one does not come close, even after 1000 samples.

Note that although the efficient algorithm is guaranteed to find a distance that is larger than or equal to the maximum distance found by the naive algorithm, unfortunate surfaces exist where neither the naive algorithm, nor the efficient algorithm are likely to find a distance close to the maximum value. This happens when the heatmap plot is fairly flat almost everywhere (i.e. almost all points in the space have similar distances from the initial point) and there is a sharp ascent of the distance function in a part of the surface having a very small area compared to the area of the entire surface. Therefore, the probability that the naive algorithm samples a point with a distance close to the maximum value or even a point whose connecting ray intersects with the part of the edge having the maximum distance is very low. Under these circumstances, none of the algorithms have a high chance of obtaining a distance close to the maximum distance.

Figure 2.23 shows the same experiment for a Boolean attribute. Since the efficient algorithm reduces the infinite sample space to the two extreme points, it guarantees to find the maximum distance with two samples. The naive algorithm might be lucky to quickly sample a good point close to the edge or might need to explore many points to get close to that maximum value.

### 2.6.3 Additional Condition Recommendation

We explore if Merlin can recommend attributes that lead to a quick result improvement (Section 2.4). The experiments compare the proposed Adaptive Tree algorithm (versions A5 and A1) against the Full Tree (F), which represents the “gold standard” in terms of producing calibrated probabilities without regard for compu-
Table 2.4: Comparison of Adaptive Tree (A5 and A1), Full Tree (F) and Random approach (R). The first row shows the initial rank and probability of the species of interest. The next rows show its state after the user added a condition for the attribute recommended by the corresponding method.

<table>
<thead>
<tr>
<th>k</th>
<th>$\rho(y)$</th>
<th>$Pr(Y \mid A_1, \ldots, A_k, D)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F</td>
<td>A5</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

a. Imprecision=0

<table>
<thead>
<tr>
<th>k</th>
<th>$\rho(y)$</th>
<th>$Pr(Y \mid A_1, \ldots, A_k, D)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F</td>
<td>A5</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>7</td>
</tr>
</tbody>
</table>

b. Imprecision=0.5

<table>
<thead>
<tr>
<th>k</th>
<th>$\rho(y)$</th>
<th>$Pr(Y \mid A_1, \ldots, A_k, D)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F</td>
<td>A5</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>7</td>
</tr>
</tbody>
</table>

c. Imprecision=1
2.6. EXPERIMENTS

tation cost. The results are also compared to a fast Random approach (R), which simply recommends a random attribute. For each attribute, we set the number of samples $s_2$ to the attribute’s domain size (see Section 2.5.1).

Table 2.4 shows a representative result for a random test record, reporting the observation of a less common, and hence difficult to identify, species initially ranked near position 100. Each row shows the rank and estimated probability of this species as a new condition is added for the $k$-th attribute selected by the corresponding algorithm.

All three tables show that Merlin (F, A5 and A1) is effective in recommending attributes that result in a quick improvement of the species’ rank (without it knowing the correct result, of course!). As expected, the tighter response time constraint for A1 leads to lower-quality probability estimates, providing faster responses but slightly poorer results. Comparing the tables from the lowest to the highest imprecision, it is also obvious that a larger fraction of imprecise conditions causes some result degradation. Still, even for Imprecision=1, the correct species quickly approaches the top-10. Note that the low probability of the target species for Imprecision=1 is not due to shortcomings of Merlin, but inherent to the problem. There are only few observations for the species, compared to those initially at the top of the list. Since an imprecise condition gives 0.25 probability mass to the wrong attribute values, this means that other species with those properties receive a significant probability mass. To explore this further, we ran another experiment where we specified all 164 attributes with imprecise responses. The maximum probability that the correct species ever reached using the Full tree was just 0.11.

Comparing these results to the Random approach, we observe that even after specifying precise conditions for 10 attributes, Random only achieves rank and probability of 67 and 0.002, respectively. For imprecise responses, Random shows no improvement in rank and probabilities. In fact, it causes a small increase in the rank of the species of interest. This shows that even a simple heuristic like entropy, which tries to select attributes that separate “winning” from “losing” species, is worth the extra computation time to help the user identify attributes that narrow the
CHAPTER 2. IMPRECISE INTERACTIVE QUERIES

2.6.4 Interactive Response Time

We explore if Merlin can indeed guarantee interactive response time. As before, the response time threshold for the Adaptive Tree is set to 5 and 1 sec for Eq. 2.10. Figures 2.24, 2.25, and 2.26 show system response time as the user adds more conditions. System response time is measured from the time the user submitted the new condition until the system responded.

Figure 2.24 shows results for a run where the user always provides precise conditions. With the Full Tree, response time initially is 544 sec, well above the threshold. (The plots cut off at 15 sec for readability.) As more conditions are added, response time decreases because more tree branches are pruned. Also, the more conditions are specified, the fewer attributes must be considered for recommending the next one. The Adaptive Tree holds response time below the threshold. As user input results in pruned tree branches, it automatically adapts the number of trees and tree levels used. In this experiment, the first attribute specified is bird size.

Figure 2.24: Response time: Imprecision = 0
with a domain size of 9. With a precise response, about $8/9$ of the tree is pruned and therefore, the time drops to about $1/9$ of the threshold. In the next round, the tree is grown to an appropriate new size, so that its response time is just below the threshold again. The Adaptive Tree ultimately converges to the Full Tree size.
The basic picture is the same in Figures 2.25 and 2.26, for scenarios with Imprecision set to 0.5 and 1, respectively. Comparing the different Full Tree curves, it is evident that response time improves more slowly for imprecise conditions. This is due to the fact that an imprecise condition does not result in pruning of tree branches as each child branch receives a non-zero weight. For Imprecision=1, no branches in the tree can be pruned. Hence Full Tree response time drops only due to the decreasing number of remaining attributes considered for future conditions.
Merlin depends on good probability estimates. We briefly motivated the use of bagged tree ensembles for probability estimation in Section 2.5.2, and provided proof of concept that they indeed work well for Merlin in practice. This chapter explores the challenge of estimating accurate, i.e., \textit{well-calibrated} probabilities, in more depth. We limit our attention to tree-based models, for which we can accurately control response time (see Section 2.5.2).

We first examine the quality of predictions by bagged tree ensembles and show that the probabilities are not well calibrated when only a small subset of the attributes is specified. Then, we show that reduced-feature models, i.e. models trained only on the attributes specified in the test record, deliver the most accurate probability estimates. This creates a tradeoff between accuracy and storage cost. A single model trained on all attributes has low storage cost but possibly poor calibration, whereas materializing a model for each possible subset of the attributes would deliver well-calibrated probabilities at very high storage cost, exponential in the number of attributes. We propose a technique that achieves nearly “the best of both worlds”, i.e. good probabilities at low storage cost. Before discussing our technique, we survey existing measures of the quality of probability estimates.
3.1 Calibration Definition

In many applications of supervised learning and in particular classification, returning the most likely class for the test instance or even a sorted list of all classes based on their likelihood is not enough and an accurate estimate of the true class membership probability is needed [51]. Measuring the quality of the class probability estimates returned by a classification model has been widely studied in machine learning. Bagged tree ensembles are shown to produce good probabilities “out of the box”, while other model types, e.g. boosting and SVMs, can also be calibrated using Platt Scaling or Isotonic Regression [39].

For Merlin, it might not be sufficient to simply rely on a single well-calibrated model trained on all attributes. The user might specify only a small subset of the attributes, therefore the model has to be well-calibrated also for such a subset. In our running example from Section 1, when Amy specifies conditions for attributes Size and WingColor, Merlin needs to predict probabilities given information on these two attributes, while hundreds of other attribute values are missing.

Measures of probability quality are based on the following intuition: Let \( S_p \) be the set of test records for which a classification model predicted that they belong to class \( Y \) with some probability “around” \( p \). For a well-calibrated model, the fraction of test records in \( S_p \) that actually belong to class \( Y \) should also be approximately \( p \) [12]. This intuition has been formalized in the following three popular calibration measures.

1. Consider a test data set of \( n \) instances. Each instance is labeled with a class attribute \( Y \in \{0,1\} \) and is assessed by the classification model with a probability \( p \in [0,1] \) where \( p \) is the probability that \( Y = 1 \). Hence, \( 1 - p \) is the probability that \( Y = 0 \). Suppose there are \( k \), \( k \leq n \), distinct probability values \( p_i, i = 1, \ldots, k \). Let \( n_i \) be the number of instances with associated probability \( p_i \). Also, let \( r_i \) be the fraction of the corresponding \( n_i \) instances whose true label is indeed 1. The Calibration or the Calibration Loss is then
3.1. CALIBRATION DEFINITION

defined to be

\[ C_n = \sum_{i=1}^{k} n_i (r_i - p_i)^2 / n \]

[6].

The intuition is to group all instances with the same predicted probability and compare their predicted value to the average of their true values. The final calibration value is the weighted average of all groups. The range of calibration is \([0, 1]\), with 0 showing perfectly calibrated probabilities and 1 showing maximum difference (of 1) between all probability estimates and their true values.

This definition has three limitations:

- It assumes that the value of \( Y \) is either 0 or 1.
- It assumes that there is a limited number of distinct prediction values, \( k \). If all predictions are different from each other, then this measure is the same as mean squared error.
- Since groups are based on equality of predicted probability estimates, a tiny change in the value of predicted probabilities would significantly affect grouping. For example, consider five instances that have an equal predicted probability of \( p = 0.8 \) and are hence, grouped together. Assume that among these five instances, four instances have an actual class value of 1 and the class value of the other one is 0. Therefore, this group is perfectly calibrated, because \( r \), the fraction of corresponding instances whose \( Y \) value is indeed 1, is also 0.8. Now changing each probability by a tiny value, e.g. 0.00001 or similar breaks the group apart into multiple poorly calibrated ones, even though the predictions are essentially the same from a practical point of view. For instance, suppose that we change the probabilities of two instances with \( Y = 1 \) to \( p_1 = 0.80001 \) and change the probabilities of the other three instances to \( p_2 = 0.79999 \). This results in two new groups with poorly calibrated probabilities, because now \( r_1 = 1 \) and \( r_2 = 0.66 \).
2. Caruana and Niculescu-Mizil used the following definition of calibration in [12]: “order all cases by their predicted value, and put cases 1-100 in the same bin. Calculate the percentage of these cases that are true positives. This approximates the true probability that these cases are positive. Then calculate the mean prediction for these cases. The absolute value of the difference between the observed frequency and the mean prediction is the calibration error for this bin. Now take cases 2-101, 3-102, . . . and compute the errors in the same way for each of these bins. CAL is the mean of these binned calibration errors.”

This approach is not well-defined for test records with the same predicted probability. Depending on their (arbitrary) re-ordering, the computed calibration value can change. For instance, consider a case where the window length is 2 and the dataset contains 6 test records: 3 with true label 1 and 3 with true label 0. Assume that the predicted value for all test records is \( p = 0.5 \). Now observe that for sort order (0, 0, 0, 1, 1, 1), the 5 windows contribute a total of 1 + 1 + 0 + 1 + 1 = 4, while for sort order (0,1,0,1,0,1), the windows contribute 0 total. Hence, the calibrated value for the former is \( 4/5 = 0.8 \) and 0 for the latter. Since there is no well-defined sort order, when two or more points have equal predicted probabilities, we pick a random sort order.

The strength of this definition is that it always looks at 100 values in a bin, from which one can reasonably estimate the average of predicted probabilities and the average of actual class values. In the first definition with equal-probability bins, some bins might have very few values, resulting in unreliable estimates of their average values.

The limitations of this approach are:

- It assumes that the value of \( Y \) is either 0 or 1.
- It introduces an ad hoc parameter which determines the length of the sliding window. The length of the window (which is set to be 100 in
this case) affects the value of Calibration. Intuitively, larger windows allow more reliable estimates of true average fraction and average predicted probability in the window, but also average “away” individual poor predictions. In one extreme, when window length is 1, calibration would be equal to absolute error, taking each individual error into account. In the other extreme, when window length is equal to the data size, it is enough for the classifier to predict the overall fraction of class 1 to be perfectly calibrated.

3. Niculescu-Mizil and Caruana used the following definition for calibration in [39]: “First, the prediction space is discretized into ten bins. Cases with predicted value between 0 and 0.1 fall in the first bin, between 0.1 and 0.2 in the second bin, etc. For each bin, the mean predicted value is plotted against the true fraction of positive cases. If the model is well calibrated the points will fall near the diagonal line.” This method is also used in Matlab for plotting the reliability diagram for calibration of a two-class predictor.

The way that this approach is introduced is only to plot a calibration graph, but not to compute the value of calibration. One can easily apply the idea of computing the error, either squared error as in the first definition or absolute error as in the second definition, and compute calibration by getting the weighted average error over all bins.

The limitations of this approach are:

- It assumes that the value of $Y$ is either 0 or 1.
- Like the first definition, there could be bins with few data points, resulting in unreliable estimates of the average probability in a bin.
- It introduces an ad hoc parameter for the number of bins. The number of bins (which is set to 10 in this case) affects the value of calibration. Its effect is similar to the effect of window length: increasing the number of bins (i.e. decreasing the width of each bin) makes individual
errors more visible; and decreasing the number of bins (i.e. increasing
their width) hides the effect of errors by computing an average over
many instances in a bin.

- The bin assignment of instances close to the bin boundaries is very
  sensitive to the exact value of probability estimates, i.e., a small change
  in predicted probability could move a record to a neighboring bin.

In our experiments, we use the second definition with sliding windows. The
main reason is that the first and third definitions suffer from unreliable estimates
of the average probability in a bin when there are too few data points in the bin.
This is not a problem for the second definition, because the sliding window always
contains exactly 100 data points.

Another reason for ruling out the first definition is that having limited number
of distinct probability values is a strong assumption for our real-world prediction
problem. In bird identification application, the number of distinct probability val-
ues returned by each individual tree is about 2000. The bagged tree ensemble
returns the average of probabilities returned by individual trees, making the overall
number of distinct probabilities much higher than that of an individual tree. Lastly,
in our problem with many missing and imprecise attribute values, the probability
returned by an individual tree is itself an average over many probabilities, making
the number of distinct probability values even higher. Having many distinct prob-
ability values makes the first definition not suitable, for two reasons: First, if there
are many distinct values, the whole idea of taking average over instances is likely
to degenerate. As individual instances fall in their own bin, calibration would turn
into mean squared error. Second, if there are millions of distinct probability values
in $[0, 1]$, those values are very close to each other, making the calibration metric
very sensitive to small changes of probabilities. In that case, the arising question is
if two probability values do not have a meaningful difference, why should they fall
in different bins.

We prefer the second definition over the third one for two reasons: First, the
third definition is based on generating equi-width bins, while the second definition generates equi-depth windows. When the bins are equi-width, bins with too few data points are possible which results in unreliable probability estimates. One could fix this with equi-depth bins, which the sliding window measure essentially does. Second, the bin in which instances that are close to the bin boundaries would fall is very sensitive to the exact value of their probability estimates. They could fall in a totally different bin with small changes in their probability values.

**Calibration for Multi-Class Problems** We stated earlier that all definitions of calibration suffer from being limited to two-class problems (i.e. binary classification). Merlin must be able to handle multi-class problems, because the attribute ranker deals with multivalued attributes and the entity ranker solves a multi-class prediction problem. For instance, the entity ranker in our bird example predicts the probability distribution over all 367 bird species in the data set.

In order to compute the calibration of a multi-class model, we propose a “one-vs.-rest” approach. For each class value, we consider that value as the positive class and all the other class values as negative. Then we compute calibration for this two-class problem.

Having $c$ class values, computing all $c$ calibration values using one-vs.-rest approach is not as expensive as $c$ independent computations of two-class calibration over the same data size. The expensive part of computing calibration is running test instances through the model to predict the class membership probabilities. Once the predicted probabilities are computed, the rest of the computation, i.e. computing per window errors, is simple and inexpensive. We divide the one-vs.-rest approach into two phases: the testing phase and the evaluation phase. The former is the expensive phase and the latter is the inexpensive one. It turns out that we need to run the test phase only once, obtaining all $c$ class values together at once, and hence, reducing the cost significantly. During the test phase, we store the predicted distribution over all class values for each test instance, along with its true class. The evaluation phase is performed for each class value separately and it starts with a pre-processing on the output of the testing phase. For each class value $y_i$, the
predicted distribution from the testing phase is converted to a distribution over two
classes: the positive class having the predicted probability of class \( y_i \) and the neg-
ative class having the sum of all other probabilities. Similarly, the true class is
replaced with 1, if the original true class is \( y_i \), otherwise it is set to 0.

Figure 3.1 illustrates our one-vs.-rest approach with a sample test data having 5
test instances, each from one of three classes \( y_1, y_2 \) and \( y_3 \). During the testing phase,
each instance is tested using model \( M \) only once and the predicted probabilities for
all classes are appended to the test instance. The output of the testing phase is
given as the input to the evaluation phase. For \( Y = y_1 \), the input is pre-processed
to replace \( y_2 \) and \( y_3 \) with a new class \( \bar{y}_1 \) which represents the negative class. Also,
the probability of the positive class is set to the predicted probability for \( y_1 \). At
this point, we have a two-class problem for which we can compute the value of
calibration. A similar process is repeated for \( y_2 \) and \( y_3 \).

**Calibration Graph:** A calibration graph illustrates the average of actual class
values versus the average of predicted class values for each bin. The bins can
either be the bins having equal probability instances (as in the first definition of
calibration) or equi-depth bins generated by the sliding window (as in the second
definition) or the equi-width bins generated by dividing \([0, 1]\) range to mutually
exclusive equal width windows (as in the third definition). For a perfectly calibrated
model, the two values are equal for all bins and the calibration graph perfectly
follows the diagonal line.

One difference between the calibration graph and the calibration value is that
in the calibration graph, the density of points in not always visible. For example,
when many points are on top of each other, they appear in the plot like a single
point. Hence, sometimes a curve does not look well calibrated, although the actual
calibration value is low, because many points are on top of each other on or near
the diagonal.
3.1. CALIBRATION DEFINITION

$$X_1 \ldots X_m Y$$

<table>
<thead>
<tr>
<th>$X_1$</th>
<th>$X_2$</th>
<th>$X_3$</th>
<th>$Y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_{11}$</td>
<td>$a_{21}$</td>
<td>$a_{31}$</td>
<td>$y_2$</td>
</tr>
<tr>
<td>$a_{12}$</td>
<td>$a_{22}$</td>
<td>$a_{32}$</td>
<td>$y_1$</td>
</tr>
<tr>
<td>$a_{13}$</td>
<td>$a_{23}$</td>
<td>$a_{33}$</td>
<td>$y_1$</td>
</tr>
<tr>
<td>$a_{14}$</td>
<td>$a_{24}$</td>
<td>$a_{34}$</td>
<td>$y_2$</td>
</tr>
<tr>
<td>$a_{15}$</td>
<td>$a_{25}$</td>
<td>$a_{35}$</td>
<td>$y_1$</td>
</tr>
</tbody>
</table>

Test Data

- **Testing Phase**
  - Run instances through model $M$ once

- **Predict probabilities**

<table>
<thead>
<tr>
<th>$X_1$</th>
<th>$X_2$</th>
<th>$X_3$</th>
<th>$Y$</th>
<th>$Pr(Y=y_1)$</th>
<th>$Pr(Y=y_2)$</th>
<th>$Pr(Y=y_3)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_{11}$</td>
<td>$a_{21}$</td>
<td>$a_{31}$</td>
<td>$y_2$</td>
<td>0.2</td>
<td>0.7</td>
<td>0.1</td>
</tr>
<tr>
<td>$a_{12}$</td>
<td>$a_{22}$</td>
<td>$a_{32}$</td>
<td>$y_1$</td>
<td>0.8</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>$a_{13}$</td>
<td>$a_{23}$</td>
<td>$a_{33}$</td>
<td>$y_1$</td>
<td>0.7</td>
<td>0.2</td>
<td>0.1</td>
</tr>
<tr>
<td>$a_{14}$</td>
<td>$a_{24}$</td>
<td>$a_{34}$</td>
<td>$y_2$</td>
<td>0.1</td>
<td>0.0</td>
<td>0.9</td>
</tr>
<tr>
<td>$a_{15}$</td>
<td>$a_{25}$</td>
<td>$a_{35}$</td>
<td>$y_1$</td>
<td>0.9</td>
<td>0.1</td>
<td>0.0</td>
</tr>
</tbody>
</table>

**Evaluation Phase**

- Compute calibration for each class

<table>
<thead>
<tr>
<th>$Y$</th>
<th>$Pr(Y=y_1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y_1$</td>
<td>0.2</td>
</tr>
<tr>
<td>$y_1$</td>
<td>0.8</td>
</tr>
<tr>
<td>$y_1$</td>
<td>0.7</td>
</tr>
<tr>
<td>$y_1$</td>
<td>0.1</td>
</tr>
<tr>
<td>$y_1$</td>
<td>0.9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$Y$</th>
<th>$Pr(Y=y_2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y_2$</td>
<td>0.7</td>
</tr>
<tr>
<td>$y_2$</td>
<td>0.1</td>
</tr>
<tr>
<td>$y_2$</td>
<td>0.2</td>
</tr>
<tr>
<td>$y_2$</td>
<td>0.0</td>
</tr>
<tr>
<td>$y_2$</td>
<td>0.1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$Y$</th>
<th>$Pr(Y=y_3)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y_3$</td>
<td>0.1</td>
</tr>
<tr>
<td>$y_3$</td>
<td>0.1</td>
</tr>
<tr>
<td>$y_3$</td>
<td>0.1</td>
</tr>
<tr>
<td>$y_3$</td>
<td>0.9</td>
</tr>
<tr>
<td>$y_3$</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Figure 3.1: One-vs.-rest approach for computing calibration in multi-class problems
CHAPTER 3. CALIBRATION AND PREDICTION QUALITY

3.2 Calibration with Partial Input

As explained earlier, one challenge that Merlin faces is that the conditions specified by the user are provided incrementally. In the beginning, none of the attribute values are known. Then in each round of interaction with the framework, the user might specify one more attribute. Therefore, the test instance for which the model predicts probability estimates is only partially specified. We use the term partial input to refer to an instance with potentially many missing attribute values. For an application like Merlin which relies on probability estimates, we would like to train well-calibrated classification models that predict highly accurate class membership probabilities, even when we have partial input and only a small subset of attribute values is known.

In Section 2.5.2 we justified why bagged tree ensembles are a good match for Merlin framework. Full bagged decision tree ensembles are an example of complete models. We use the term complete model to refer to any prediction model trained on all attributes \( X_1, \ldots, X_m \in D \). When Merlin uses a full bagged tree ensemble for prediction, many attribute values will be unknown and hence treated as missing values at prediction time. Our Full bagged tree ensemble consists of decision trees implemented based on Quinlan’s C4.5 [42]. These trees handle missing values using the distribution-based imputation [44].

Bagged tree ensembles are known to predict well calibrated probability estimates [39]. But those results were shown for complete input test instances, i.e. test instances that are not partial. Figure 3.2 illustrates this concept for our full bagged tree ensemble when there is no missing value in the input. All experimental results in this chapter are obtained with the same setup as in Section 2.6. The green dotted line in this graph is a calibration curve, based on the predicted probabilities for a sample of 10000 test instances. The dashed black line is the diagonal line, plotted as a reference to better illustrate the distance between the calibration curve and the optimal calibration. Recall that the diagonal line represents perfect calibration, while significant deviations indicate poor probability estimates. The calibration is
3.2. CALIBRATION WITH PARTIAL INPUT

Figure 3.2: Calibration graph for full bagged tree ensemble with all known attribute values evaluated using the one-vs.-rest approach introduced in Section 3.1. The positive class in this graph is the Tree Swallow and all other classes are mapped to the negative class. The calibration curve follows the diagonal line very closely, showing that the full bagged tree ensemble indeed returns well calibrated probability estimates. The value of calibration for this sample is $1.641 \times 10^{-4}$.

Based on this observation and the fact that full bagged tree ensembles predict well calibrated probability estimates, a natural approach to achieve well calibrated probabilities for partial input instances is to train and use a designated reduced-feature model for each input. For example, when Size and WingColor are the only specified attributes, we should ideally use a model trained specifically on these two
attributes.

Next, we compare the calibration of the complete model against that of reduced-feature models for different subsets of attributes.

### 3.2.1 Complete and Reduced-Feature Models

When using the terms complete model and reduced-feature models, we are specifically referring to bagged tree ensembles, the former being the full bagged tree ensemble trained on all attributes (from Section 2.5.2) and the latter being the reduced-feature bagged tree ensemble trained only on the specified attributes.

In this section, we show experimental results for comparison between the calibration of the complete model and the reduced-feature models. The general procedure in these experiments is to select a subset of attributes as the specified attributes. All the other attributes would be set to missing in the test instances. Then the test instances having missing values are once tested with the complete model and another time with a reduced-feature model particularly trained on the specified subset of attributes.

In decision trees, attributes are not equally important and their values do not have the same effect on the quality of the predicted probability estimates. Intuitively, attributes that appear closer to the root of the trees have stronger effect on the final prediction, making early decisions about what branches to follow. On the other hand, attributes that appear in the lower levels of the tree, i.e. closer to the leaves, have only local effects on the remaining instances in corresponding branches. Based on this observation, we examine the effect of attributes from the top and the bottom levels of the trees separately. Figure 3.3 illustrates examples of top and low attributes in a sample tree.

**Top attributes:** We explored the structure of the complete model and found an ordered list of the top 16 attributes, occurring most often in the top levels of the trees. Then, we selected the subset of specified attributes from these 16 attributes: In the initial round nothing was specified, and in each round one attribute was added
3.2. CALIBRATION WITH PARTIAL INPUT

to the subset of known attributes. Here, we present the results for 0, 1, 2, 4, 8, and 16 known attributes. The calibration values are shown in Table 3.1. Figure 3.4 illustrates the calibration graphs for 0, 1 and 2 attributes for complete and reduced-feature models. Figure 3.5 illustrates similar graphs for 4, 8 and 16 attributes.

In Table 3.1, $k$ represents the number of specified attributes, $\text{CAL}_C$ shows the calibration of the complete model and $\text{CAL}_R$ shows the calibration of the reduced-feature models. The table shows that $\text{CAL}_R$ is always smaller than $\text{CAL}_C$, meaning that reduced-feature models generate more calibrated probability estimates than the complete models. Note that due to artifacts of the data or the model induction process, in practice it is not guaranteed that $\text{CAL}_R$ will always be better. However, as a tendency, complete model is expected to be poorly calibrated, because it makes decisions on unspecified attributes.

Although, the value of calibration in Table 3.1 does not necessarily decrease as more attributes are specified, the calibration graphs in Figures 3.4 and 3.5 clearly illustrate the effect of specifying more attributes. Figures 3.4a and 3.4b show the calibration curves for complete and reduced-feature models respectively, when no attribute is specified. With all attributes missing, the complete model distributes all test instances in the same way through the split nodes and hence, predicts the same probability for all of them. The single point on the $x$-axis, i.e. 0.08, shows the
Figure 3.4: Comparison between calibration curves of complete model and reduced-feature models when 0, 1 and 2 top attributes appearing in the complete model are selected as the specified attributes.
3.2. CALIBRATION WITH PARTIAL INPUT

Figure 3.5: Comparison between calibration curves of complete model and reduced-feature models when 4, 8 and 16 top attributes appearing in the complete model are selected as the specified attributes.
Table 3.1: Comparison between calibration values of complete model and reduced-feature models when the top attributes appearing in the complete model are selected as the specified attributes.

<table>
<thead>
<tr>
<th>$k$</th>
<th>$\text{CAL}_C$</th>
<th>$\text{CAL}_R$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0223</td>
<td>0.0222</td>
</tr>
<tr>
<td>1</td>
<td>0.0154</td>
<td>0.0135</td>
</tr>
<tr>
<td>2</td>
<td>0.0337</td>
<td>0.0035</td>
</tr>
<tr>
<td>4</td>
<td>0.0241</td>
<td>0.0030</td>
</tr>
<tr>
<td>8</td>
<td>0.0016</td>
<td>0.0015</td>
</tr>
<tr>
<td>16</td>
<td>0.0024</td>
<td>0.0011</td>
</tr>
</tbody>
</table>

overall frequency of the positive class. The points on the $y$-axis are distributed in a neighborhood around 0.08, because all points have equal predicted probabilities and as explained in Section 3.1, they are randomly shuffled, resulting in bins having different, but almost equal, average of actual values. Essentially the vertical line shows the variance of the number of positive instances across the bins.

The same pattern is observed for the reduced-feature model. The only difference is that the reduced-feature model does not need to distribute the test instances through a large tree to get the overall frequencies. These frequencies are stored in the single leaf of this model.

Figures 3.4c and 3.4d show the calibration curves after the first attribute is specified. This attribute, located at the root node of the complete model is Size with 9 values. In both figures, four vertical lines are clearly visible. This is due to the fact that the size of the individual birds from the species which is mapped to the positive class, i.e. Tree Swallow, can be one of four possible values 1, 2, 3 or 4. In Figure 3.4c for the complete model, the other points are achieved in two possible ways: 1. At a split point with a missing value in the test instance, the instance is distributed to all branches, getting a weighted average of the probabilities over all leaves it reaches. 2. Assume a group of instances having the same predicted probability $p_1$ and another group with probability $p_2$, such that $p_1 < p_2$ and there is no other predicted probability between them. With the sliding window approach, the window initially contains instances from the first group and as it slides towards larger probability values, instances of the first group are removed gradually from
3.2. CALIBRATION WITH PARTIAL INPUT

the sliding window and instances from the second group enter the window. This results in intermediate average probabilities along the $x$-axis. In Figure 3.4d for the reduced-feature model, the latter is the only cause for getting points other than the points on the four vertical lines.

Figures 3.4e and 3.4f show the calibration curves after the second attribute is specified. It can be observed that the points on the curves are more distributed and less centered around few predicted probabilities. Another interesting pattern that can be observed in these two figures and becomes clearer in Figure 3.5 with more attributes specified, is the migration of a dense group of points towards point $(1, 1)$. These points are records that belong to the positive class. As more attributes are specified, the capability of the models in predicting a true value of 1 for these rare points increases. Figure 3.5f best illustrates the dense point at $(1, 1)$ containing instances from the positive class.

To summarize, when the specified attributes are selected from the top parts of the complete tree, the reduced-feature model predicts more calibrated probability estimates, but the difference compared to the reduced-feature models is not significant in some cases, e.g. when the root of the complete model is known, the predicted probabilities by the models are almost the same. Next, we conduct similar experiments, but this time we select attributes from lower parts of the complete tree.

**Low attributes:** By exploring the structure of the complete model, we selected 4 attributes that do not appear in the top 4 levels, but appear either in level 5 or 6. The calibration values are shown in Table 3.2. Figure 3.6 illustrates the calibration graphs when 1, 2 and 4 attributes are specified.

Table 3.2 shows that $\text{CAL}_R$ is always smaller than $\text{CAL}_C$, meaning that reduced-feature models generate better calibrated probability estimates than the complete models. Unlike the calibration values in Table 3.1 for the top attributes, the difference between the calibrations of the two models is now significant and meaningful. The reason is that the low attributes in the complete model do not have a significant effect on the overall probability estimates and are only able to affect a
Figure 3.6: Comparison between calibration curves of complete model and reduced-feature models when 1, 2 and 4 low attributes are selected as the specified attributes.
Table 3.2: Comparison between calibration values of complete model and reduced-feature models when four low attributes are selected as the specified attributes.

<table>
<thead>
<tr>
<th>$k$</th>
<th>$\text{CAL}_C$</th>
<th>$\text{CAL}_R$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0237</td>
<td>0.0220</td>
</tr>
<tr>
<td>1</td>
<td>0.0590</td>
<td>0.0208</td>
</tr>
<tr>
<td>2</td>
<td>0.0588</td>
<td>0.0149</td>
</tr>
<tr>
<td>4</td>
<td>0.0619</td>
<td>0.0123</td>
</tr>
</tbody>
</table>

few instances. But in the reduced-feature model, these attributes have a significant effect, because they are the only attributes in the tree.

Figure 3.6 shows the same pattern as in Figures 3.4 and 3.5: The reduced-feature model better follows the diagonal line compared to the complete model. It also shows that the complete model now suffers more from the partial instances, because the specified attributes are not in the top part of the tree. Another interesting pattern observed in these calibration graphs is that the reduced-feature model with 4 low attributes, shown in Figure 3.6f, is not as good as the reduced-feature model with 4 top attributes, shown in Figure 3.5b. The reason is that the low attribute, even when positioned at the root of the tree, is not capable of making “pure” splits.

In this section, we showed that reduced-feature models generate better calibrated probability estimates than the complete models. We also showed that this effect is stronger when the known attribute values do not appear in the top part of the complete model. Based on this observation, we discuss the positive and negative points of using a complete model in Merlin. Then we propose possible solutions to achieve better calibrated probabilities.

### 3.2.2 Calibrated Probabilities in Merlin

We conducted experiments in Section 3.2 to show that complete models are not as good as reduced-feature models in terms of generating calibrated probabilities. But we also showed that the difference between the quality of their predicted probabilities is significant if the specified attributes do not appear in the top part of the trees in the complete model. Based on these observations, we can conclude that Mer-
lin does not suffer from this problem as long as the user specifies the top-ranked attributes suggested by Merlin. Although this might often be the case and the complete model used by Merlin is shown to have a very good performance in practice (see Section 2.6), three issues still threaten complete models:

1. Although Merlin suggests a ranked list of attributes to the user, the user might not have or want to provide the information for the top attributes recommended by Merlin. In that case, the attribute chosen by the user might be a low attribute.

2. Consider two highly correlated attributes \(X_1\) and \(X_2\). During the tree induction, if the tree splits on \(X_1\), then the information gain (or other similar measures) of splitting on \(X_2\) would be low, because a split on \(X_2\) would be similar to a split on \(X_1\). Therefore, it is unlikely to use \(X_2\) in any subtree rooted at \(X_1\). However, if the user at runtime specifies \(X_2\) and not \(X_1\), then the tree cannot make use of this information, because no node splits on it.

3. Recall from Section 2.5.2 that satisfying an interactive response time is one of the requirements of Merlin and therefore, we introduced the adaptive tree model as a limited version of the complete model. The tree might split on attribute \(X\), but if the node is below the limit, then it does not affect the prediction process, i.e., is equivalent to the case of it not being in the tree (or subtree of interest).

These issues are addressed by reduced-feature models, because they are trained only on the specified attributes. Since Merlin must guarantee interactive response time, training a new model on-the-fly for each new subset of specified attributes will usually not be an option, except for very small data sets. On the other hand, with hundreds of attributes, pre-training all possible combinations of specified attributes is not an option due to storage cost, even though it might meet the interactive response time requirements (Assuming that fetching the models can be done quickly). To combine the best of both worlds, i.e., the well-calibrated probabilities
of reduced-feature models and the small storage cost of a single complete model, we propose a new layer model. Layered models are trained on the reduced set of features, but rely on a simpler training procedure to enable interactive training on-the-fly.

### 3.2.3 Layered Models

The idea behind layered models is similar to lazy tree induction algorithms [23]. In these algorithms, the tree induction phase is delayed until a test instance is given, at which point, the tree is specifically trained for the given test instance. At each level, the best split node and the matching branch for the test instance must be found. In our case, the process is simpler, because the split attribute is determined by the attribute for which the user just added a new condition.

The layered model has a tree structure in which the split nodes are the specified attributes in order. We always add the latest specified attribute as the split node to the last level of the tree. This eliminates the expensive tree induction cost of the reduced-feature models.

Test instances with precise conditions always follow a single path in the tree, making it sufficient for the induction algorithm to grow that path only. In our application, since we allow imprecise conditions, the test instance might follow multiple paths. Moreover, since we allow users to modify conditions, we always grow the full tree.

When the user specifies an attribute, the split node for the next level is determined. Then the branches under the split node are generated and the corresponding instances in the training data are assigned to the branches. At the same time, the class membership probabilities are computed by counting instances in each obtained leaf. Figure 3.7 illustrates this process when the user specifies attribute $X_3$. In this example, the user has first specified attribute $X_4$ and then $X_1$ in the previous rounds and the layered model is already trained on these two attributes. Figure 3.7.a shows the current layered model trained on attributes $X_4$ and $X_1$. The array
CHAPTER 3. CALIBRATION AND PREDICTION QUALITY

Figure 3.7: Layered model induction.
3.2. CALIBRATION WITH PARTIAL INPUT

structure shown below the tree contains all training instances. Instances in the sub-
array with endpoints \( n_i \) and \( n_{i+1} \) correspond to the leaf with the range \([n_i, n_{i+1})\). One training instance is shown as an example. In this instance, \( X_4 \) is equal to 1 and \( X_1 \) is equal to \( N \). When we run the test instance through the model, it fol-
lows the path ending in the leaf with range \([0, n_1)\). As illustrated in the picture, it is also in the subarray with endpoints 0 and \( n_1 \) in the array. When attribute \( X_3 \) is specified, it must be added to the next level under all four leaves. Figure 3.7.b shows the first leaf and its corresponding subarray that is going to be expanded by adding attribute \( X_3 \) in the next level under it. As Figure 3.7.c illustrates, we execute a function called \( \text{sortAndSplit} \) on this leaf to add \( X_3 \) to the next level. Algorithm 3 shows the pseudocode of \( \text{sortAndSplit} \). This function takes a tree node \( \Delta \) with its range \([lo, hi)\) and the splitting attribute \( X \) as the input and returns a set of leaves with their ranges to be added as the children of the input node \( \Delta \). The first step is to sort the instances in the subarray having the range \([lo, hi)\) on attribute \( X \). Then for each value \( x \in X \), it creates a new leaf. In order to find the ranges of these leaves, it needs to scan the subarray to find positions where the value of \( X \) changes. Although finding the endpoints can be done faster using a modified bi-
nary search algorithm, scanning the array is necessary, because at the same time we also check the entities that the instances belong to and count the frequency of each entity. This produces the predicted probability estimates for each new leaf. The result of calling this function on the first leaf is illustrated in Figure 3.7.c. Note that the sample training instance illustrated in Figure 3.7.a is moved to the correct range in the array during the sorting phase. Figure 3.7.d shows the final layered model having all three attributes and its corresponding array.

Before we present experimental results, we need to discuss three concerns about the layered models: exponential growth, training time and testing time.

**Exponential growth:** Looking at the layered model as a tree structure, we add the last specified attribute to the last level in all branches. When \( k \) Boolean attributes are specified, the number of leaves in the tree is \( 2^k \). This would potentially result in an exponential growth of the tree. Fortunately, the number of leaves is
bounded by the size of the training data, because when we split on many attributes, we might end up with leaves having a single training instance. At that point, no further splits can be done. Moreover, we might choose to stop splitting when the number of instances in a node is less than a predefined constant. This avoids overfitting and results in more reliable predictions.

**Training time:** When adding a new attribute to the layered model, the `sortAndSplit` function is called on all current leaves. This function a) sorts the subarray that corresponds to the input leaf on the new attribute and b) finds the ranges of the newly generated leaves and computes the entity probabilities for each leaf. Here, we investigate the time needed for each component:

The time needed for sorting an array of size $m$ is $O(m \log m)$. When the `sortAndSplit` function is called on all leaves, all subarrays in the array are sorted locally. The computational complexity of sorting all sub-partitions of an array locally is at most as high as the complexity of sorting the entire array. (Consider an array of size $m_1 + m_2$ containing two subarrays of sizes $m_1$ and $m_2$. Now observe that $O((m_1 + m_2) \log (m_1 + m_2)) = O(m_1 \log (m_1 + m_2) + m_2 \log (m_1 + m_2)) \geq O(m_1 \log m_1) + O(m_2 \log m_2))$. For a training data having $n$ training instances, the cost of the sorting phase is at most $O(n \log n)$. In our experiments, we use a training data of 2 million instances. On the machine that we use for experiments, the sorting takes less than 70 milliseconds.

The time needed for finding the ranges of the new leaves and computing the
entity probabilities is linear in the size of the subarray corresponding to the parent node. We scan the sorted instances in the subarray and for each instance, we a) check the value of the splitting attribute and b) check the entity. The former is used to find the ranges of the new leaves, because a change in the value of the splitting attribute determines the upper bound for one leaf and the lower bound for another. The latter is used to compute the entity probabilities for each leaf. This process is done for all leaves, therefore the overall time is bounded by $n$, the size of the training data.

Putting it all together, the time required for adding a new attribute to the layered model is $n \log n + n + u$ where $u$ is the overhead for creating the nodes in the tree. In our experiments with 2 million training instances, the entire process of adding the latest specified attribute to the last level of the tree takes less than 110 milliseconds.

**Testing time:** Testing in the layered model comes almost for free. When `sortAndSplit` is called on a node that the test instance has reached, the predictions are directly read from the returned leaves. The only additional cost is to find the matching leaves among the returned leaves.

Now we present experimental results similar to those presented for complete and reduced-feature models in Section 3.2.1. Again, we explore the effect of top and low attributes, but this time when using the layered model for probability estimation.

**Top attributes:** We specified the 16 top attributes, one by one, in the same order as in Section 3.2.1. The calibration values are presented in Table 3.3. In order to make comparison against complete and reduced-feature models easier, we include the calibration values of those models from Table 3.1 in this table. CAL$L$ represents the calibration value for the Layered model. Figure 3.8 illustrates the calibration graphs for 0, 1, 2, 3, 4 and 5 attributes. Figure 3.9 illustrates similar graphs for 6, 8, 10, 12, 14 and 16 attributes.

Table 3.3 clearly shows that for all values of $k$, except for $k = 16$, the reduced-feature model is the most calibrated model among the three models compared. The next most calibrated model is the layered model having calibration values very
Figure 3.8: Calibration curves of layered model when 0, 1, 2, 3, 4 and 5 top attributes appearing in the complete model are selected as the specified attributes.
Figure 3.9: Calibration curves of layered model when 6, 8, 10, 12, 14 and 16 top attributes appearing in the complete model are selected as the specified attributes.
Table 3.3: Comparison between calibration values of complete model, reduced-feature models and layered model when the top attributes appearing in the complete model are selected as the specified attributes.

<table>
<thead>
<tr>
<th>k</th>
<th>CAL&lt;sub&gt;C&lt;/sub&gt;</th>
<th>CAL&lt;sub&gt;R&lt;/sub&gt;</th>
<th>CAL&lt;sub&gt;L&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0223</td>
<td>0.0222</td>
<td>0.0222</td>
</tr>
<tr>
<td>1</td>
<td>0.0154</td>
<td>0.0135</td>
<td>0.0137</td>
</tr>
<tr>
<td>2</td>
<td>0.0337</td>
<td>0.0035</td>
<td>0.0054</td>
</tr>
<tr>
<td>4</td>
<td>0.0241</td>
<td>0.0030</td>
<td>0.0036</td>
</tr>
<tr>
<td>8</td>
<td>0.0016</td>
<td>0.0015</td>
<td>0.0015</td>
</tr>
<tr>
<td>16</td>
<td>0.0024</td>
<td>0.0011</td>
<td>0.0009</td>
</tr>
</tbody>
</table>

close to those of the reduced-feature models. The worst model in terms of calibration is the complete model with values far from those of reduced-feature and the layered models. For \( k = 16 \), the layered model produces slightly more calibrated probabilities.

Figures 3.8 and 3.9 show that as more attributes are specified, the layered model returns more calibrated probabilities. When 6 attributes are specified, the calibration graph is already very close to the diagonal line.

This experiment confirms that the layered model returns well-calibrated probabilities. We also showed before that the training time that the layered model takes is very low. Therefore, the layered model indeed achieves nearly “the best of both worlds”. Next, we consider the effect of specifying low attributes.

**Low attributes:** We specified the 4 low attributes, one by one, in the same order as in Section 3.2.1. The calibration values are presented in Table 3.4. Again, we include the calibration values of complete and reduced-feature models from Table 3.2 in this table to make comparison easier. Figure 3.10 illustrates the calibration graphs for 1, 2, 3 and 4 attributes.

Table 3.4 shows the same pattern as in Table 3.3: The less calibrated model is always the complete model and the reduced-feature model returns better calibrated probabilities than the layered model. For \( k = 2 \), the layered model is slightly better than the reduced-feature model.

To summarize, we introduced layered models and showed that they return well-calibrated probabilities like the reduced-feature models, but have the low storage
### 3.2. CALIBRATION WITH PARTIAL INPUT

![Calibration curves of layered model](image)

(a) Layered model, 1 attribute  
(b) Layered model, 2 attributes  
(c) Layered model, 3 attributes  
(d) Layered model, 4 attributes

Figure 3.10: Calibration curves of layered model when 1, 2, 3 and 4 low attributes are selected as the specified attributes.

Table 3.4: Comparison between calibration values of complete model, reduced-feature models and layered model when the low attributes are selected as the specified attributes.

<table>
<thead>
<tr>
<th>$k$</th>
<th>$\text{CAL}_C$</th>
<th>$\text{CAL}_R$</th>
<th>$\text{CAL}_L$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0237</td>
<td>0.0220</td>
<td>0.0222</td>
</tr>
<tr>
<td>1</td>
<td>0.0590</td>
<td>0.0208</td>
<td>0.0229</td>
</tr>
<tr>
<td>2</td>
<td>0.0588</td>
<td>0.0149</td>
<td>0.0148</td>
</tr>
<tr>
<td>4</td>
<td>0.0619</td>
<td>0.0123</td>
<td>0.0126</td>
</tr>
</tbody>
</table>
cost (single model) and interactive response-time properties of the complete model.

To show that these results are not only correct for the eBird data set, next we show that similar results are achieved for another data set as well.

### 3.2.4 Experiments

We repeated the calibration experiments for Adult (Census Income) data set, a data set from UCI Machine Learning Repository [30].

#### 3.2.4.1 Data

Adult data set is used to predict if annual income exceeds $50K based on census data. Adult contains 48842 records, each described with 14 numerical and categorical attributes. Since in exploratory search continuous domains can be discretized (see Section 2.1.3), we converted numerical attributes to categorical attributes using an equi-depth histogram with 10 bins. The class attribute has two values: “$\leq$ $50K”” and “$>$ $50K$”. The calibration results are shown for class “$\leq$ $50K$”.

#### 3.2.4.2 Top Attributes

Since the Adult data set has only 14 attributes, we show calibration results for the top 4 attributes. The effect of adding the next top attributes was not significant. Figure 3.11 illustrates the calibration graphs for 0 and 1 attributes for complete and reduced-feature models and Figure 3.12 illustrates similar graphs for 2 and 4 attributes. Figure 3.13 shows the calibration graphs for layered models trained on the top 0, 1, 2 and 4 attributes.

The graphs show that the results obtained from the experiments on the eBird data set carry over to the Adult data set as well. Comparing the calibration graphs of complete and reduced-feature models in Figures 3.11 and 3.12 we observe that the reduced-feature models are more calibrated than the complete model. Initially, when no attribute is specified, the single point on the $x$-axis is 0.76 which is the overall frequency of the positive class. As more attributes are specified the calibra-
3.2. CALIBRATION WITH PARTIAL INPUT

Figure 3.11: Comparison between calibration curves of complete model and reduced-feature models when 0 and 1 top attributes appearing in the complete model are selected as the specified attributes in Adult data set.

Figure 3.13 shows that the layered models are not as well-calibrated as the reduced-feature models, but are better than the complete model. This is particularly observable in the calibration curves corresponding to 2 and 4 attributes. For instance, comparing Figure 3.12c to Figure 3.13d, we observe that the complete
model makes more overestimations or underestimations of the true probabilities than the layered model. Also, a comparison between Figure 3.12b to Figure 3.13c shows that the calibration curve of the layered model does not follow the diagonal line as closely as the curve of the reduced-feature model does.

These findings are also confirmed by the calibration values in Table 3.5. The calibration value of the layered model is always between those of complete and reduced-feature models.
Figure 3.13: Calibration curves of layered model when 0, 1, 2 and 4 top attributes appearing in the complete model are selected as the specified attributes in Adult data set.

Table 3.5: Comparison between calibration values of complete model, reduced-feature models and layered model when the top attributes are selected as the specified attributes in Adult data set.

<table>
<thead>
<tr>
<th>$k$</th>
<th>$\text{CAL}_C$</th>
<th>$\text{CAL}_R$</th>
<th>$\text{CAL}_L$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0362</td>
<td>0.0333</td>
<td>0.0335</td>
</tr>
<tr>
<td>1</td>
<td>0.0285</td>
<td>0.0276</td>
<td>0.0281</td>
</tr>
<tr>
<td>2</td>
<td>0.0314</td>
<td>0.0277</td>
<td>0.0288</td>
</tr>
<tr>
<td>4</td>
<td>0.0306</td>
<td>0.0293</td>
<td>0.0299</td>
</tr>
</tbody>
</table>
Table 3.6: Comparison between calibration values of complete model, reduced-feature models and layered model when the low attributes are selected as the specified attributes in Adult data set.

<table>
<thead>
<tr>
<th>k</th>
<th>CAL_C</th>
<th>CAL_R</th>
<th>CAL_L</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0362</td>
<td>0.0336</td>
<td>0.0338</td>
</tr>
<tr>
<td>1</td>
<td>0.0362</td>
<td>0.0334</td>
<td>0.0335</td>
</tr>
<tr>
<td>2</td>
<td>0.0362</td>
<td>0.0314</td>
<td>0.0347</td>
</tr>
<tr>
<td>4</td>
<td>0.0363</td>
<td>0.0311</td>
<td>0.0346</td>
</tr>
</tbody>
</table>

3.2.4.3 Low Attributes

We selected 4 attributes appearing in the lower levels of the complete tree as low attributes. Hence, it is guaranteed that the low attributes appear somewhere in the complete model, but their effect on the predicted probabilities might not be significant.

Figure 3.14 illustrates the calibration graphs for 0 and 1 low attributes for complete and reduced-feature models and Figure 3.15 illustrates similar graphs for 2 and 4 low attributes. Figure 3.16 shows the calibration graphs for layered models trained on the low 0, 1, 2 and 4 attributes.

Again, we observe a similar pattern here: The reduced-feature models are more calibrated than the layered models and the complete models are not able to compete with them, since the low attributes have almost no effect on their predictions. The calibration values, presented in Table 3.6, clearly confirm our conclusions.
3.2. **CALIBRATION WITH PARTIAL INPUT**

Figure 3.14: Comparison between calibration curves of complete model and reduced-feature models when 0 and 1 low attributes appearing in the complete model are selected as the specified attributes in Adult data set.
Figure 3.15: Comparison between calibration curves of complete model and reduced-feature models when 2 and 4 low attributes appearing in the complete model are selected as the specified attributes in Adult data set.
3.2. CALIBRATION WITH PARTIAL INPUT

Figure 3.16: Calibration curves of layered model when 0, 1, 2 and 4 low attributes appearing in the complete model are selected as the specified attributes in Adult data set.
Interactive exploratory search requires an acceptable response time which is not a universal constant, but depends on the user’s inquiry, preferences, and environment [33]. To the best of our knowledge, Merlin is the only system for imprecise queries where the user can set system response time according to her preferences, gracefully trading accuracy of estimated probabilities for faster response time. BlinkDB [2] takes a different approach toward interactive query exploration in parallel systems. It trades off result accuracy for faster response time by running a (precise) query on an appropriate data sample.

Query steering [13] is a general framework for interactive data exploration. Merlin proposes concrete solutions for the interactive performance (Section 2.5) and navigation help (Section 2.4) components of this framework. Previous work on query steering and interactive query refinement requires the user to provide positive or negative examples of desired results. For example, in the AIDE data exploration framework [21] the user marks objects as relevant and irrelevant. Then a decision tree is trained based on this information on all already marked objects. The predicates of the areas of interest are then specified from the tree. The goal is to maximize F-measure of the final decision tree. The phases include 1) relevant object discovery where objects are sampled from a hierarchical exploration grid and shown to user for relevance feedback, 2) misclassified exploitation where the framework tries to sample more relevant objects in areas around false negative objects to learn more relevant areas, 3) boundary exploitation where the framework
selects more objects in distance $x$ of the current boundaries to shrink/expand the boundaries based on user’s relevance feedback on those objects. The query refinement framework of Islam et al. [26] requires the user to specify missing and undesirable result tuples. Using the point domination theory and by capturing user’s intent, the framework derives other implicit expected and unexpected results based on user’s feedback. Next step is to find the optimal set of conditions to be tightened or relaxed to capture the unexpected and expected results. This problem is solved in a greedy manner. Abouzied et al. [1] propose techniques for learning quantified Boolean queries by asking users to label data objects as answers or non-answers. Given a set of missing result tuples, ConQueR [47] automatically modifies an SQL query so that it contains both the original result and the missing tuples, minimizing addition of other tuples. The crucial assumption underlying these approaches is that the user can explicitly state desired result tuples. Hence they are not applicable to Merlin’s target applications where the user cannot provide the results, but is searching for them by specifying (possibly imprecise) conditions.

Relaxation was explored for queries that return an empty result. Mottin et al. [37] propose an interactive approach where the user is guided towards a desirable relaxation. Similarly, Junker [28] explores the space of possible relaxations for interactive applications based on user preferences. For Select-Project-Join queries, the Stretch’n’Shrink technique [34] interactively finds relaxations and contractions based on user feedback in order for the query to reach a target output cardinality. The main idea is to first find the maximal transformation on each dimension (i.e. the maximal relaxation or contraction needed to achieve the cardinality of interest if only that dimension is refined). With a user in a loop, in each step the user refines one predicate (by specifying a point in the range between the current value and the maximal transformation value) and then the framework updates the maximal transformation for other predicates that are not refined yet. For $d$ dimension, this procedure is repeated $d - 1$ times and the last dimension is automatically fixed by the framework to satisfy the cardinality constraint. All these approaches are concerned with navigating a huge space of possible query relaxations or contractions.
based on user preferences about which conditions are changeable. This is orthogonal to our work, where the goal is not to automatically modify a condition, but to efficiently process imprecise conditions and analyze their impact on the result.

The general idea of selecting good questions to efficiently extract information from a user has appeared in various contexts. The 20Q game [10] uses a proprietary algorithm to select good questions in order to guess an object the user is thinking of. It mixes ideas from artificial neural networks and binary search for selecting the best objects and the next question. Visipedia [8] determines the category of an image by posing questions to the user based on visual properties. A similar problem of selecting questions and their order was also explored in the context of questionnaire design [16]. The notion of revealing information at a cost has also been explored in active learning [45]. The most common scenario is as follows: Given a set of input vectors and a budget, choose some inputs to be labeled with their class such that a model trained on the resulting labeled data set has the highest accuracy. Work on active feature acquisition and classification considers input attribute values to be revealed at a cost and tries to balance this cost with the benefit of correctly classifying a given input [24, 27]. These approaches can in principle be incorporated into the Merlin framework for ranking of unspecified attributes. However, none of them considers tuneable response time. The general idea of reducing user-effort in interactive systems also appeared in other contexts, e.g., ontology matching [18] and constraint-based query systems for pattern discovery [7]. For human-assisted graph search, Parameswaran et al. [40] explore how to select an optimal set of graph nodes to minimize the number of reachability questions a human expert has to answer.

While also helpful for composing a query, query suggestion and auto-completion are orthogonal to our problem. They rely on repositories of historic queries to predict what query a user might be interested in. One study shows that although a short query prefix does not give good results in query auto-completion, considering the context, i.e. recent queries, and suggesting queries that are most similar to the context improves the results significantly [5]. In another study, the
immediate preceding queries are considered as context. This approach is called context-aware query suggestion. The approach consists of two steps: in the offline step, similar queries are grouped together to form concepts and in the online step, the concept of the queries in the context is found and queries are suggested based on the concept [11]. In another work query autocompletion is studied in the presence of typing errors [15].

Research on probabilistic databases [46] concentrated on dealing with imprecise data, while Merlin’s design is centered around imprecise queries. The theory community explored search in the presence of errors, which considers correct and incorrect, but not imprecise user input. Finocchi et al. [22] study the problem of computing with unreliable information and name the models that have been widely explored, among which, some variations of the “liar model” are the closest to our problem, except that imprecise responses are not studied. Not dealing with imprecise responses, allows finding an upper bound on the number of questions needed to determine a value of interest when the search involves linearly bounded erroneous answers [4]. Another main difference between most studies in this field and Merlin is about the type of questions. While Merlin asks questions about the properties of entities (i.e., attribute values), the questions studied by the theory community are mainly membership questions. Putting it together, these studies solve a very different problem and neither Merlin can solve the problems in the theory community, nor those studies are able to replace Merlin. For example, a Merlin user who is trying to find an entity of interest is not able to answer membership questions, otherwise the entity was already known! Also, not being able to provide imprecise responses, the entity of interest might get eliminated due to an incorrect response. On the other hand, supporting imprecise responses in a probabilistic framework, prevents Merlin from analyzing the number of needed questions for finding the result of interest. In another survey, variations of the “twenty questions” are studied where responses are either correct or incorrect [29].

Previous work on imprecise queries focused on finding result tuples “similar to” a desired one. For example, in [36], vague queries are defined in contrast to specific
queries as queries that establish a target qualification and are concerned with data close to this target. Vague queries allow users to find their results of interest without repeatedly trying alternative queries with small changes in the values. A vague query contains a vague selection comparator, called “similar-to”, that returns any data within a predefined distance of the specified value. AIMQ [38] handles an imprecise query by turning it into a set of precise (equality) queries, then finds the most relevant tuples in a neighborhood based on provided or inferred distance measures. This notion of query imprecision is fundamentally different from ours. Our goal is to find the precise result the user is looking for. In the bird example, the user wants to identify the actual species, not others “like it”. Furthermore, our framework is designed to let the user express uncertainty of the type “most likely the bird had blue on the wing, but there is a small chance it did not.” These types of statements are easy to express with probability distributions, but cannot be captured by the notion of “similar” results, because it does not make sense to consider having wing color blue to be similar to not having wing color blue. However, it might be interesting to combine the two approaches so that the user can start with imprecise conditions (using Merlin), and then, after finding a match, expand the search to similar entities to double-check for possible mis-identification.

Agrawal et al. [3] are among the first to explore ranking of results for database queries. Given a SQL query on a database system, with empty answers or many answers, it is desirable that the database returns a ranked list of approximately or best matching tuples. The authors apply the idea of TF/IDF from information retrieval to the problem by penalizing more frequent attribute values over all tuples as they are less informative and distinctive. Recent work includes [14] where the authors propose ranking functions to rank the results of a many-answers query by looking at the unspecified attributes. An unspecified attribute might be important because of its high global score which captures its global importance, or because of its high conditional score which captures the strengths of dependencies between specified and unspecified attributes. Since all these approaches deal with precise queries, a tuple is either in the query result or not. Hence ranking is based on
additional properties, e.g., frequency of attribute values or global importance of unspecified attributes. Uncertainty arises not from the query, but for example when the importance of an unspecified attribute is estimated based on specified ones. In Merlin, imprecise queries lead to a natural notion of ranking based on probability distributions of specified attributes. In addition to result ranking, for databases with many attributes the notion of attribute ranking was studied [19]. The authors propose that when a data set has many attributes, displaying the results of a query to the user, only a subset of attributes can be displayed. Instead of a predefined or manual selection of attributes, an automatic selection based on their influence on the ranking is desired. Attribute ranking could be added to Merlin for result presentation.

While we proposed methods to train well-calibrated models, some studies explore techniques to post-process the predicted probabilities of models, e.g. naive bayes and boosting models, to obtain well-calibrated predictions [39, 51]. Caruana and Niculescu-Mizil [12] compare nine classification performance metrics, including calibration and show that calibration is very different from other metrics, except for squared error and cross entropy. The relationship between calibration and squared error is also shown in the decomposition of the Brier Score [20].

Saar-Tsechansky and Provost [44] explored methods for handling missing values at prediction time. Discarding instances, acquiring missing values, imputation and using reduced-feature models are the four approaches for handling missing values. Among these approaches, the first two are not considered, because depending on the problem, they might not always be applicable. For the other two, they show that reduced-feature models outperform imputation techniques. In this work, we explore the same techniques, but in terms of their calibration properties, not other evaluation metrics, e.g. accuracy. As expected, our results agree with that previous work in the sense that reduced-feature models come out as the winner also for calibration. In addition to exploring the effect of missing values on calibration, we also proposed layered models to get the calibration quality of reduced-feature models, without having to pre-compute and manage the exponential number of possible
reduced-feature models.

Classification is a well-studied problem in data mining and machine learning [25]. The goal is to learn a model to predict the class label for a given input. Issues related to interactive refinement of model input are not considered, but we can leverage traditional classification techniques for probability estimation as discussed in Section 2.5.

Search and ranking suggest a relationship to information retrieval [31], but our work deals with structured data. We can leverage research on measures for similarity and distance of ranked result sets for our proposed notion of sensitivity, as discussed in Section 2.3.6.

Crowd-sourcing refers to a variety of approaches that involve humans in the creation of data and the solution of problems that are difficult for computers [32, 49]. Merlin provides novel imprecise search capability for non-technical users to access crowd-sourced databases.
CHAPTER 5

Conclusions and Future Work

To gain wider acceptance for big data analysis and exploratory search, databases have to support a broad spectrum of users in finding the information they are looking for. We proposed Merlin as a novel framework to allow users to interactively compose imprecise queries by explicitly expressing the uncertainty through probabilistic conditions.

We introduced the desired functionalities of Merlin and provided analytical and experimental results to show how Merlin supports them. The probabilistic framework is the core to all functionalities supported by Merlin. This framework is needed, because imprecision can be present both in the data and the user-provided query conditions.

One of the functionalities supported by Merlin is to provide constant feedback to the user, presenting a ranked list of the entities based on their likelihood of being the entity of interest, given the specified conditions by the user. We showed how to rank the entities to minimize the expected user effort.

We proposed the novel notion of sensitivity to help user understand the potential risk of specifying a condition that she is not certain about. Knowing how sensitive the resulting ranked list is to the value of the condition, the user can decide if she wants to modify or eliminate the condition. We proved an important monotonicity property and proposed an algorithm that exploits it for more efficient estimation of sensitivity by “sampling from the edge”. Our results also indicate that it is un-
likely for a more efficient general sensitivity computation algorithm to exist. Using experimental results we showed how our efficient algorithm works in practice.

Another functionality supported by Merlin is to recommend additional conditions to the user based on their potential for improving the quality of the ranked list of entities. The user can go through the list and choose one of the top attributes that she is confident to specify a condition for. We showed experimental results as a proof of concept that the conditions recommended by Merlin indeed improve the quality of predicted probability estimates, causing the correct entity to appear near the top of the ranked list of entities after only a few conditions are specified.

For Merlin to be interactive, it has to respond within a time the user finds acceptable. Merlin allows the users to pick their personal preferred response time threshold. Since estimating probabilities is the bottleneck of all processes performed by Merlin, we proposed adaptive trees as a solution for fast probability estimation. Adaptive trees are designed to adapt their size to the given response time threshold by automatically tuning the number of accessed levels or the number of accessed individual trees in their ensemble. Experimental results show that the adaptive trees are very close to the full trees in terms of the quality of predictions.

In the last part of this dissertation, we investigated the calibration of the probabilities predicted by Merlin and established the need for reduced-feature models to achieve well-calibrated probabilities. To avoid excessive storage cost and unacceptable on-the-fly training time of reduced-feature models, we proposed layered models that are only trained on the specified attributes. Experimental results showed that these models are well-calibrated and can be trained incrementally on-the-fly, without jeopardizing interactive response time.

In this dissertation, we chose bagged tree ensembles for probability estimations and justified why they match the requirements of Merlin. We also conducted experiments to show that the quality of the predictions is indeed good. Exploring other approaches for fast and accurate probability estimation is an interesting direction for future work.

In all the experiments, we always assumed that the user-provided conditions are
present in the form of a probability distribution. In practice users would express their uncertainty using suitable interfaces. It would be interesting to explore such interfaces and to study how to turn the user-provided input into the probability distributions that Merlin is designed to work with.

We conducted experiments to show that reduced-feature models are more calibrated than the complete models. An analytical model to explore this finding is a possible future work in this field.
Bibliography


