UC Berkeley
UC Berkeley Electronic Theses and Dissertations

Title
High-Performance Systems for Crowdsourced Data Analysis

Permalink
https://escholarship.org/uc/item/8ch3q8gj

Author
Haas, Daniel

Publication Date
2017

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

High-Performance Systems for Crowdsourced Data Analysis

by

Daniel Haas

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Michael J. Franklin, Co-chair
Professor Kenneth Goldberg, Co-chair
Professor Thomas L. Griffiths

Summer 2017

High-Performance Systems for Crowdsourced Data Analysis

Copyright 2017
by
Daniel Haas

Abstract

High-Performance Systems for Crowdsourced Data Analysis

by

Daniel Haas

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Michael J. Franklin, Co-chair

Professor Kenneth Goldberg, Co-chair

In spite of the dramatic recent progress in automated techniques for computer vision and natural language understanding, human effort, often in the form of crowd workers recruited on marketplaces such as Amazon's Mechanical Turk, remains a necessary part of data analysis workflows for machine learning and data cleaning. However, embedding manual steps in automated workflows comes with a performance cost, since humans seldom process data at the speed of computers. In order to rapidly iterate between hypotheses and evidence, data analysts need tools that can provide human processing at close to machine latencies.

In this dissertation, I describe the design, theory, and implementation of performant crowd-powered systems. After discussing the performance implications of involving humans in data analysis workflows, I present an example of a data cleaning system that requires low-latency crowd input. Then, I describe CLAMShell, a system that accurately labels large-scale datasets in one to two minutes, and its evaluation on over a thousand workers processing nearly a quarter million tasks. Next, I consider the design of multi-tenant crowd systems running many heterogeneous applications at once. I describe Cioppino, a system designed to improve throughput and reduce cost in this setting, while taking into account worker preferences. Finally, I explore the theory of identifying fast individuals in an unknown population of workers, which can be modeled as an instance of the infinite-armed bandit problem. The analysis results in novel near-optimal algorithms with applications to broader statistical theory. Together, these components provide for the implementation of human computation systems that are cost-efficient, scalable, and fast enough to integrate into existing data analysis workflows without compromising performance.

To the crowds who support me.

Contents

Contents ii

List of Figures vi

List of Tables viii

1 Introduction 1
- 1.1 Algorithms, Machines, and People . . . 2
- 1.2 Where do Humans Fit in Data Analysis? . . . 3
- 1.3 Not as Easy as it Sounds . . . 5
- 1.4 Optimizing Crowds by Analogy . . . 6
- 1.5 Thesis Overview and Contributions . . . 6
- 1.6 Organization . . . 7

2 Background 9
- 2.1 The Cast of Characters . . . 9
- 2.2 Human Computation Platforms . . . 10
- 2.3 Human-in-the-Loop Data Analysis . . . 12
  - 2.3.1 Overview and Example Application . . . 12
  - 2.3.2 Selecting Records to Process . . . 14
  - 2.3.3 Synthesizing Task Interfaces . . . 14
  - 2.3.4 Recruiting Workers . . . 15
  - 2.3.5 A Sample of Applications . . . 16
- 2.4 Challenges in Human-Powered System Design . . . 17
  - 2.4.1 Quality Control . . . 17
  - 2.4.2 Cost . . . 18
  - 2.4.3 Performance . . . 19
  - 2.4.4 A Focus on Latency . . . 20
- 2.5 Other Related Work . . . 23
  - 2.5.1 Declarative Systems and Database Optimization . . . 23
  - 2.5.2 Distributed Cluster Computation . . . 24
  - 2.5.3 Cloud Infrastructure Management . . . 25

2.5.4 Multi-Armed Bandits 25

3 Wisteria: An Interactive Data Cleaning System 27
3.1 Introduction 27
3.2 System Architecture 30
3.2.1 Architecture Overview 30
3.2.2 Cleaning DSL 31
3.2.3 Cleaning Operators 31
3.3 Planning Interface 32
3.3.1 Plan-Level View 32
3.3.2 Operator-Level View 32
3.4 Supporting Interactive Plan Generation 33
3.4.1 Sampling 33
3.4.2 Recommendation 34
3.4.3 Crowdsourcing 35
3.5 Conclusion 35

4 CLAMShell: Speeding up Crowds for Low-Latency Data Labeling 36
4.1 Introduction 36
4.2 Tackling Crowd Latency 37
4.2.1 Existing Literature 38
4.2.2 Towards a Comprehensive Solution 39
4.3 The CLAMShell System 39
4.4 Per-batch Latency Optimization 41
4.4.1 Straggler Mitigation: Reducing Variance 42
4.4.2 Pool Maintenance: Better Mean Latency 44
4.4.3 Combining Per-Batch Techniques 45
4.5 Full-Run Latency Optimization 47
4.5.1 Hybrid learning 47
4.5.2 Active learning batch size 48
4.5.3 Active learning decision latency 49
4.5.4 Putting it all together 49
4.6 Evaluation 49
4.6.1 Experimental Setup 50
4.6.2 Pool Maintenance 51
4.6.3 Straggler Mitigation 55
4.6.4 Combining Per-Batch Techniques 56
4.6.5 Hybrid Learning 59
4.6.6 End-to-End Evaluation 61
4.7 Conclusion 63

5 Cioppino: Multi-Tenant Crowd Management 64

5.1 Introduction 65
5.2 Related Work 66
5.3 The Cioppino System 67
5.3.1 Cioppino: a multi-tenant crowd system 67
5.3.2 System model 69
5.4 Pool Elasticity 69
5.4.1 Optimal queue size 69
5.4.2 Autoscaling algorithms 70
5.4.3 Cooling period 72
5.5 Pool Stability 72
5.5.1 Worker abandonment 73
5.5.2 Pool stability 73
5.5.3 Interaction with pool elasticity 74
5.6 Pool Balance 74
5.6.1 Approach overview 74
5.6.2 Estimating queue completion time 75
5.6.3 Evaluating candidate transfer sets 76
5.6.4 Searching the transfer set space 77
5.7 Evaluation 77
5.7.1 Experimental Setup 77
5.7.2 Pool elasticity 78
5.7.3 Pool stability 80
5.7.4 Pool balance 82
5.7.5 End-to-end evaluation 84
5.8 Conclusion 87

6 Identifying Fast Workers in Unknown Populations 88
6.1 Introduction 88
6.1.1 The Most Biased Coin Problem 89
6.1.2 Related Work 90
6.1.3 Problem Statement 92
6.2 From Identifying Coins to Detecting Mixture Distributions 93
6.3 Lower bounds 94
6.3.1 Adaptive strategies 95
6.3.2 The detection of a mixture distribution and the most biased coin problem 95
6.3.2.1 Sample complexity when parameters are known 96
6.3.2.2 Sample complexity when parameters are unknown 97
6.4 A Discussion of Gaussians 98
6.4.1 On the detection of a mixture of Gaussians 98
6.4.2 Lower bounds 99
6.4.3 Gaussian Upper bound for known $\alpha, \theta_0, \theta_1$ 99
6.5 Near optimal adaptive algorithm 100

6.6 Other Upper Bounds . . . 102
6.6.1 Fixed sample size strategy . . . 103
6.6.2 Either α or ϵ is unknown, but not both . . . 103
6.7 Conclusion . . . 104

7 Future Work and Conclusions 105
7.1 Summary of Contributions . . . 105
7.2 Limitations . . . 106
7.3 Future Work . . . 107
7.4 Final Remarks . . . 108

Bibliography 110

A Proofs of Chapter 6 Theorems 120
A.1 Proof of Theorem 2 . . . 120
A.2 Proof of Corollary 1 . . . 120
A.3 Proof of Theorem 3 . . . 121
A.4 Proof of Corollary 2 . . . 124
A.5 Proof of Theorem 4 . . . 130
A.6 Proof of Theorem 5 . . . 131
A.7 Proof of Theorem 6 . . . 136
A.8 Proof of Theorem 7 . . . 138
A.9 Proof of Theorem 8 . . . 139
A.10 Proof of Theorem 9 . . . 140

List of Figures

1.1 The process of data analysis ([61]). 4

2.1 The flow of data through a human-powered data analysis system. 13
2.2 The flow of data through a human-powered data analysis system shown in Figure 2.1, with sources of latency labeled. 21

3.1 Example iterations on the design of the portion of a cleaning plan that extracts restaurant addresses from their unstructured webpages. 1) An exploratory plan that uses a sample to evaluate a simple address extraction method. 2) A plan that applies the method to the entire dataset. The quality is unsatisfactory. 3) An alternate plan that uses manual crowd extraction. The quality is now high, but the crowd-based extractor is slow. 4) A hybrid plan that sends only difficult webpages to the crowd, maximizing accuracy without sacrificing latency. 28
3.2 Wisteria system architecture, with an example entity resolution plan. 30
3.3 Wisteria's plan-level view. 32
3.4 Wisteria's operator-level view. 33

4.1 CLAMShell architecture diagram. 40
4.2 Distribution of worker latencies. 42
4.3 # points labeled over time in 2 typical runs. 51
4.4 Summary of end-to-end cost and latency experiments with and without pool maintenance. 52
4.5 Comparison between age of the worker in the pool when starting a given task and the time to complete the task. Tasks where the latency per labeled point is greater than 8 seconds are colored in blue. 52
4.6 Mean pool latency over time. 53
4.7 The number of workers replaced over time for varying maintenance latency thresholds. 54
4.8 50th, 95th, and 99th percentiles of task latency as maintenance latency threshold varies. Each facet is a different amount of time into the experiment. 54
4.9 Straggler mitigation dramatically reduces the standard deviation of per-task latency across batches. 56

4.10 Points labeled over time with straggler mitigation in 2 typical runs. 57
4.11 Straggler mitigation increases costs by 1 to 2×, improves latency by $2.5 - 5\times$, and variance by $4 - 14\times$. 57
4.12 End-to-end Latency, Variance, and Costs for different straggler mitigation and pool maintenance configurations. 58
4.13 Per-assignment view of each straggler mitigation and churn configuration. Each horizontal segment is the length of an assignment. Red and blue dots denote batch boundaries 58
4.14 Replacement rate when using `TermEst` (Section 4.4.3) with $\alpha = 1$. 59
4.15 Active, Passive, and Hybrid strategies for learning on crowds run on generated datasets in the simulator. 60
4.16 Active, Passive, and Hybrid strategies for learning on crowds run on real-world datasets on live workers. 61
4.17 Summary of end to end time to reach models of varying accuracy 62
4.18 Wall clock time vs Model Accuracy 62

5.1 Cioppino architecture diagram. 68
5.2 Control system block diagram. 71
5.3 Response of pool elasticity algorithms to workload decrease or increase, averaged over 50 runs. The workload change begins at the 400 second mark. 78
5.4 Pool stability techniques and the effects of the parameters α (the probability that each worker abandons the pool after completing a task) and r (the mean recruitment rate, $\frac{1}{r}$ is the average delay before a recruited worker arrives), averaged over 50 runs. The chart for $\alpha = 0.5, r = 0.02$ is omitted because the y axis would be scaled too high as a result of over-recruitment by AR and HR. 81
5.5 Throughput and cost of pool stability algorithms for varied α and r, averaged over 50 runs. 82
5.6 Performance of pool balance algorithms on the 3-application system faceted by tr_i, averaged over 50 runs. 83
5.7 Simulated end to end system comparison, averaged over 50 runs. 85
5.8 End to end system comparison, executed against the CLAMShell trace. 86

List of Tables

1.1 A sample of API calls provided by Amazon Mechanical Turk [7]. 3

4.1 Classification of sources of latency in data labeling. 38
4.2 CLAMShell techniques (AL: Active Learning). 49
4.3 Experimental Parameters 50

5.1 Summary of techniques used by Cioppino. 67
5.2 Summary of pool elasticity techniques evaluated in Section 5.7.2. 79

6.1 Upper and lower bounds on the expected sample complexity of different δ-probably correct strategies. Fixed refers to the strategy of Corollary 1. This table assumes $\min\{\theta_0(1-\theta_0), \theta_1(1-\theta_1)\}$ is lower bounded by a constant (e.g. $\theta_0, \theta_1 \in [1/8, 7/8]$) and $\epsilon = \theta_1 - \theta_0$ is sufficiently small. Also note that the upper bounds apply to distributions supported on $[0, 1]$, not just coins. All results without bracketed citations were unknown prior to this work. † Due to the discouraging lower bound for any estimate-then-explore strategy, it is inadvisable to propose an algorithm. 100

Acknowledgments

Of all the words in this thesis, the following were easiest to write: this document would not exist without the tireless support of my mentors, collaborators, friends, and family.

My advisor, Mike Franklin, has a knack for asking exactly the question I'm hoping nobody will. Coincidentally, thinking through that question has consistently been the most direct path towards the next deep research idea. Mike pressed me from the beginning to focus on ideas that matter, and I am profoundly grateful for his guidance. Professor Eugene Wu helped (and mocked) me through the most difficult experiments of my thesis, and should be excited to be in the paragraph about professors. Professors Jiannan Wang and Benjamin Recht have been invaluable collaborators, and my ideas were shaped by insightful conversations with Professors Joe Hellerstein, Joey Gonzalez, Ken Goldberg, Aditya Parameswaran, Arnab Nandi, and Tom Griffiths.

The incredibly bright and caring people who make up the AMPLab, Berkeley EECS, and the broader academic CS community have enriched my time in graduate school significantly. Kevin Jamieson and Sanjay Krishnan have been supportive collaborators and friends. Evan Sparks married a wonderful woman, produced a perfect child, and sometimes was fun to hang out with. Shivaram Venkataraman produced a perfect palak paneer and made a stellar brewbot partner. David Schinazi convinced me to act foolishly on several occasions. Dominic Labanowski, Tom Zajdel, and Rachel Traylor gave me a second home in Berkeley. Lydia Gu, Nitesh Banta, Jason Ansel, and the rest of my Locu and B12 families helped me publish my first paper and got me excited about the future. Adam Marcus inspired me to want to change the world with my research, and taught me that unrelenting kindness makes sarcasm even better. In various ways, the feedback, support, and beers I received from (in no particular order) Chris Thompson, Thomas Rembert, François Belletti, Ashia Wilson, Becca Roelofs, Haoyuan Li, Peter Bailis, Beth Trushkowsky, Tim Kraska, Sara Alspaugh, Dan Crankshaw, Stephen Tu, Jodi Loo, Ginger Smith, Kay Ousterhout, Daniel Bruckner, Wenting Zheng, Jonathan Harper, Josh Rosen, Neeraja Yadwadkar, and the JASONS made even the hardest moments of graduate school worthwhile. Special thanks to Margo Seltzer for showing me what research looks like in the first place.

I am additionally grateful for the unfailing positivity and occasional rule-bending of deparment staff: Jon Kuroda, Audrey Sillers, Kattt Atchley, Carlyn Chinan, and Boban Zarkovich.

Finally, my family put up with me for years before graduate school, and their love and selflessness got me through it. My parents, Peter and Laura Haas, talked about research when I wanted to and bought me dinner when I didn't, and made countless drives up to Berkeley to help me move apartments, take me for walks, and shower me with love. My brother, Joshua Haas, enthusiastically failed to read the paper I sent him, but made for great conversations as he tackled new technical challenges of his own.

Saving the best for last (as she would want), I owe pretty much all of my success in life to my wife, Aleksandra Kuczmarska-Haas. She encouraged me to apply to Ph.D. programs, patiently put up with our frustrating cross-country romance for 4 years, pushed me to work

harder when I needed to, and cuddled me on the couch when I couldn't. She also earned an MD, married me, moved across the country, bought a house with me, and consistently enveloped me in her trademark brand of warm, intelligent, fiercely loyal, ambitious love.

This research has been supported in part by an NSF graduate fellowship under Grant No. DGE 1106400, DHS Award HSHQDC-16-3-00083, NSF CISE Expeditions Award CCF-1139158, DOE Award SN10040 DE-SC0012463, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, IBM, SAP, The Thomas and Stacey Siebel Foundation, Apple Inc., Arimo, Blue Goji, Bosch, Cisco, Cray, Cloudera, Ericsson, Facebook, Fujitsu, HP, Huawei, Intel, Microsoft, Mitre, Pivotal, Samsung, Schlumberger, Splunk, State Farm and VMware.

Chapter 1

Introduction

With every passing year, technology is becoming more sophisticated and more ever-present in our lives. Our cellphones help us navigate with realtime traffic alerts and information about nearby businesses, while the software we use every day recommends the products we buy, the movies we watch, and even the clothes we wear. Many of these advances can be ascribed to what the popular media calls "The Big Data Revolution" [114], which can be summarized as the timely intersection of three trends: a vast increase in the amount of data being collected and made available over the internet, a dramatic decrease in the cost of large-scale computing mainly due to the advent of public cloud computing providers, and the development of powerful open-source software tools for computing on large datasets.

Applying massive computing resources to massive amounts of data comes with challenges, however, and though machine learning and data mining approaches have produced an impressive array of futuristic applications, there are many problems for which automated solutions fall short. For example, the best algorithms we have today cannot with high accuracy understand the emotional sentiment of tweets, detect offensive content in videos, or identify duplicate products in large datasets. These applications are critical to support the technologies that power our lives, and they all have one important feature in common: people can do them without much effort at all. Because of our innate human ability to understand language and recognize patterns, most of us could look at a tweet and decide in seconds whether it describes a positive or negative sentiment towards its subject, or examine two smartphone listings from an online marketplace and decide whether or not they are duplicates.

Taking advantage of this insight, today's companies spend millions of dollars per year paying human workers to complete simple tasks that cannot be automated [100], a practice known as "human computation". Unfortunately, putting humans into the loop between data

and insight represents a major obstacle to performance[1], since people and machines process data on very different timescales. As a result, human computation can currently support only those applications where speed is not a requirement, a shrinking set as consumers demand insights faster and faster.

In this thesis, I study techniques that bring the performance of human computation to near-machine timescales so that it can be usefully embedded in big data applications. I argue that many of the sources of latency in human-powered systems are not fundamental, and present systems [65, 66, 64] and algorithms [78] I have designed to eliminate them. These techniques manage human workforces with software in novel ways, bringing together ideas from many related areas in computer systems, computer theory, and human-computer interaction.

1.1 Algorithms, Machines, and People

The rise of crowdsourcing has been an important catalyst in the development of human computation systems. Crowdsourcing (see [50, 117] for recent surveys) has become an umbrella term for any number of techniques that allow individuals or organizations to incentivize and coordinate other people to provide services, ideas, or insights, mainly over the internet. In this thesis, I focus on *paid microwork*, a form of crowdsourcing where workers accept monetary compensation in exchange for performing small, repetitive data processing tasks such as labeling images or translating text. Microwork platforms are vital to human computation systems, which frequently break datasets into small groups of items and send them to paid workers for processing.

Though there are many public platforms for paid crowdsourcing (see Section 2.2), the most popular of them is Amazon's Mechanical Turk (MTurk) [105], which was one of the first platforms to develop programmatic application programming interfaces (APIs) on top of its crowdsourcing product. MTurk's APIs allow users to write software that automatically creates a task, hires a worker to work on the task, displays an interface to the worker within which to perform the task, pays the worker after the task is completed, and sends the results back to the software, all without any manual intervention by the software programmer. Table 1.1 lists some of the functionality provided by MTurk's APIs, which can be invoked from code written in 9 different programming languages including Python, Java, Javascript, and C++.

Such APIs provide unprecedented control over human work, and open the door for large scale hybrid human-machine computer systems where the flow of insights between people and algorithms is bi-directional. For example, adaptive algorithms can request information

[1] The systems research community uses the term *performance* to refer to data processing speed, whereas the machine learning research community uses the term to refer to the quality or accuracy of predictions. In this thesis, which is primarily focused on system design, I use the word performance to refer exclusively to system speed, and will use the terms *quality* or *accuracy* to describe the predictive power of machine learning models.

API Call	Important Params	Description
`CreateHIT`	interface specification, reward, number of answers desired.	creates a new HIT (Human Intelligence Task) and makes it available to Workers.
`CreateQualificationType`	test questions and answer key.	creates a new Qualification task that workers must pass before working on HITs.
`ListAssignmentsForHIT`	HIT identifier.	retrieves results for completed task assignments.
`ApproveAssignment`	assignment identifier.	approves the results of a completed assignment and pays the worker.
`RejectAssignment`	assignment identifier, explanatory message.	rejects a completed assignment and does not pay the worker.
`NotifyWorkers`	message text, worker identifiers.	sends an email to workers.
`SendBonus`	worker identifier, amount, reason.	issues a bonus payment to a worker.

Table 1.1: A sample of API calls provided by Amazon Mechanical Turk [7].

in the form of labeled data from workers in order to refine machine learning models, and humans can run algorithms to summarize, recommend, and visualize data to improve their understanding of it. These systems are vital to modern data-driven applications that require computational power that can only be provided by large clusters of computers, and accuracy that can only be provided by human workers. As a result, many companies report using crowdsourced human computation for training machine learning algorithms, cleaning large datasets, and moderating content [100].

1.2 Where do Humans Fit in Data Analysis?

In the abstract, it is not difficult to imagine a system where humans and algorithms pass data back and forth. But what do data analysis systems actually do, and how do they make use of the crowd? Figure 1.1 lays out the phases of modern data analysis (for a detailed description of the process, see, e.g., [48]). First, data are *acquired* from sensors, users on the internet, or existing publicly available sources. Then, data are *pre-processed* to be ready for

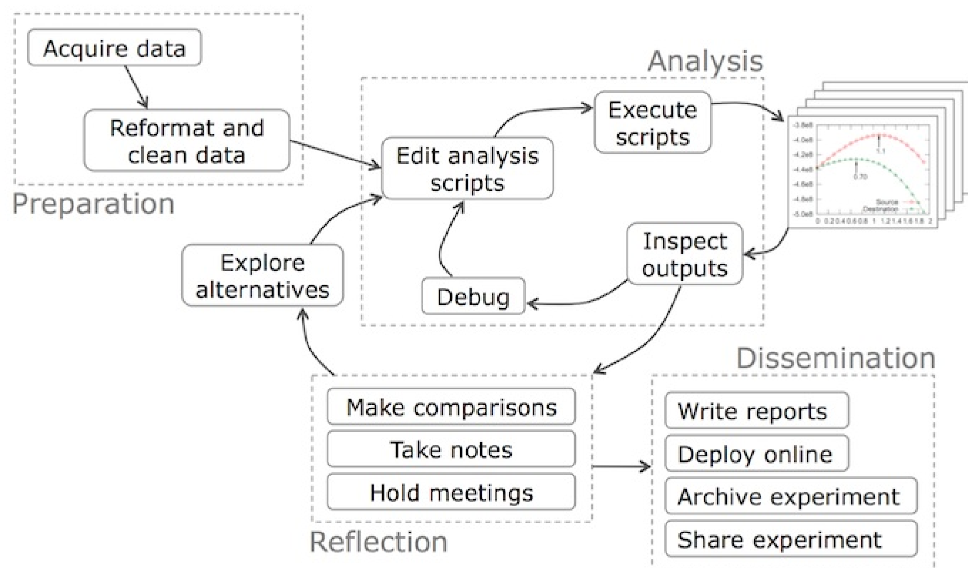


Figure 1.1: The process of data analysis ([61]).

analysis, for example by transforming fields into a consistent format or extracting structured information from text, audio, or visual data. Next, the data are *cleaned* to correct errors, fill in missing values, and eliminate duplicate entries. Subsequently, data analysts perform *exploratory data analysis* by computing descriptive statistics of the data, querying the data, or generating visualizations of the data. This step frequently leads to observations that require further investigation by acquiring new data or adjusting pre-processing and cleaning techniques. Finally, *predictive models or algorithms* are trained on the data to generate insights, for example recommendation algorithms based on product data, or topic clustering based on textual data. These algorithms, once deemed to be of sufficient quality, are built into a data product used to make the decisions that power businesses and support user applications.

Data science is typically performed by individuals or small teams within modern companies, but crowd workers are increasingly being used to aid in data analysis during all of its phases. MTurk is frequently used as a data collection tool for survey data [15], and crowd systems for data acquisition can augment existing datasets by asking the crowd to, e.g., list all of the computer science professors at Berkeley [137]. In the pre-processing phase, human-in-the-loop systems allow workers to reformat data entries and guide algorithms for automatic reformatting [82], as well as performing complex data extraction from images or pdfs [63]. In the data cleaning phase, crowds can assist in acquiring missing data [112], identifying duplicate data [58], and resolving challenging data errors [34] (the system presented in Chapter 3 is an example of a system that can do all three). During exploratory data analysis, crowds can help design useful visualizations [26] or execute open-ended queries that require human interpretation of the data [52, 102, 110]. And to support predictive models,

crowds are frequently tasked with providing labeled training data to improve the accuracy of the models, for example by labeling objects in images [45] or translating text [23].

With APIs like those of MTurk, software systems can automatically hire crowd workers and assign them data analysis work instead of requiring data scientists to create crowd tasks by hand. These systems constitute an additional layer of abstraction on top of the crowd platform, allowing data scientists to specify high level goals such as 'eliminate duplicates in this dataset of products' rather than writing low-level code to hire and send tasks to workers. In addition to improving usability, the extra abstraction allows systems to optimize how they gather human input without affecting the data scientist's specification of analysis goals. This is analogous to relational database systems, which optimize how data is accessed independently of how users specify SQL queries. Consequently, the optimization techniques presented in this thesis can improve the performance of crowd systems without changing how data analysts ask for crowd input.

1.3 Not as Easy as it Sounds

Unfortunately, designing human-powered systems comes with a unique set of challenges, because human processors are different from computers in crucial ways. People have varying levels of skill and experience, and do not complete tasks the same way every time, making unintentional errors due to lack of attention or fatigue. They are orders of magnitude slower at processing a single data item than a computer, and the time they take to complete a tasks varies significantly over time and between individuals. They have inherent motivations and interests that lead them to perform with different accuracy and speed on different tasks. Ignoring these differences can lead to systems that counterintuitively produce results of lower accuracy than purely automated systems, while simultaneously costing more and taking longer to compute those results.

As a result, existing human computation systems explicitly model human behaviors and design techniques to address them, focusing on improving one or more of three key goals of a human computation system: reducing *cost*, increasing *quality*, and maximizing *performance*. These goals are all highly interrelated. For example, techniques that reduce cost by reducing the fraction of data processed by human workers [143, 101, 111] can hurt accuracy because answers require approximation, but will improve performance because there is less work to do. Techniques that attempt to improve quality by getting more opinions [75, 18], on the other hand, will increase costs because more workers must be paid, and will hurt performance because getting more opinions takes longer.

Much of the previous research on optimizing crowd systems has focused on tradeoffs involving cost and quality, allowing performance to fall by the wayside. However, supporting modern interactive data analysis workflows also requires a focus on crowd performance. Waiting days for the crowd to process a large dataset is unacceptable when the analysis workflow is being re-adjusted every few minutes, or when the data is constantly changing.

In this thesis, I focus on optimizing crowd performance in order to design systems that can process large datasets in minutes, not days.

Intuitively, addressing crowd performance seems like a fundamental challenge due to the difference in timescales between human and machine processing. However, people can complete individual tasks such as image labeling in less than a second, and automated data analysis systems routinely take minutes to hours to train machine learning models or query large datasets. One of the key arguments of this thesis is that by identifying and eliminating the bottlenecks that slow crowd systems down, crowd-powered data analysis can complete on timescales similar to those of completely automated systems.

1.4 Optimizing Crowds by Analogy

Ironically, although the major challenges to the design of crowd-powered systems arise from the behaviors that set people apart from computers, many of the system optimizations presented in this thesis and elsewhere that address those challenges are inspired by the similarities between human and machine. For example, representing crowd tasks as functions that take an input and produce a structured output allow for optimizations that collect multiple outputs for the same task and average them together.

A key analogy that I leverage in this thesis is between crowds of human workers and clusters of distributed computers. In many ways, sending data processing tasks to multiple people and gathering their responses is like a distributed system whose master server coordinates computation across numerous worker computers, especially those that do so in multiple stages, such as mapreduce systems [43] or distributed dataflow engines [147]. Similarly, supporting multiple user applications running on a single crowd is analogous to multi-tenant cloud computing environments that must share resources fairly but efficiently between the clients in the system. These analogies are useful because, unlike in crowd systems, an immense amount of effort has been focused on optimizing the performance of distributed computation, both in the systems and theory communities. In this thesis, I take inspiration from the analogy between crowd workers and computers, and adapt systems techniques such as straggler mitigation [10], scheduling [108], autoscaling [116], and multi-tenant resource sharing [56], as well as the learning theory of multi-armed bandits [77] to help improve the performance of humans in supporting data analysis.

1.5 Thesis Overview and Contributions

This thesis focuses on designing systems and algorithms to accelerate crowd-powered systems to near-machine performance so they can be embedded in automated data analysis workflows. In it, I make the following contributions [65, 66, 78, 64]:

- I motivate performant crowd systems with examples of modern data analysis systems that rely on repeated low-latency input from humans. I have released Sample-

Clean [124], one such system, as open source software.

- I analyze in depth the sources of latency in crowd systems, and use that understanding to identify where effort should be focused in order to improve crowd performance.
- I leverage the similarities between crowd data analysis and distributed cluster computation to bring theory and techniques from other areas to bear on crowd performance.
- Building on ideas from these related areas, I propose optimizations and algorithms that dramatically reduce latency, increase throughput, and lower the cost of crowd-powered data analysis systems. My proposals explicitly take into account the subtleties of human behavior that can affect performance, and support multiple applications running on a single performant crowd. Many of these techniques are available as part of the open-source software package AMPCrowd [9].
- I evaluate these proposals extensively via three main mechanisms: large-scale experiments on live crowd workers, simulations that use synthetic and trace-based task workloads and crowd behavior to explore how environmental variation affects the impact of techniques, and proofs that offer high-probability guarantees on the performance of algorithms. The CLAMShell system presented in Chapter 4 can reduce the latency of crowd-powered data labeling by up to 8× and reduce its variance by up to 150×, and the Cioppino system presented in Chapter 5 can reduce the cost of data analysis on a multi-tenant crowd by up to 19× while increasing throughput by 20%.

The ultimate goal of this thesis is to support a new class of system that can improve quality by integrating people into the data analysis process without significantly altering the end-to-end performance of automated data analysis systems. This is only possible because of the separation between the level of abstraction at which system users specify the data analysis tasks they want humans to work on, and the level of abstraction at which the system hires, pays, and assigns tasks to crowd workers. By hiding the complexity of performant crowdsourcing from data analysts and taking advantage of low-level APIs for crowd work, I allow crowd-powered systems to implement optimizations that improve performance without affecting the way in which data and results are communicated.

1.6 Organization

The remainder of this thesis is organized as follows. Chapter 2 introduces crowd-powered systems for data analysis, with a focus on the obstacles to achieving near-machine performance. In doing so, it surveys related work in crowd-powered systems as well as the several other areas on which this thesis draws.

Chapter 3 further motivates performant crowd-powered systems with an in-depth look at Wisteria, an example of the kind of human-in-the-loop system that is enabled by performant crowds. Wisteria allows users to specify plans for data cleaning with either automated or

human operators, and brings them together under a unified workflow abstraction. To make this abstraction work, the system relies on crowd operators running at comparable timescales to automated operators, and requires consistency in crowd response times.

Chapter 4 demonstrates that consistent performance from crowds is possible, and that latency can be reduced to the point where large datasets can be processed by humans in minutes. I describe CLAMShell, a system that incorporates multiple optimizations designed to reduce the task, batch, and full-run latency of crowd-powered data analysis. These optimizations are inspired by techniques from cluster computing, and include straggler mitigation, pool maintenance, and hybrid active-and-passive learning. I evaluate CLAMShell on a crowd of over 1,000 MTurk workers running nearly 250,000 tasks, showing order of magnitude reductions in the speed and variance of getting answers from the crowd.

In Chapter 5, I extend CLAMShell with additional techniques for managing pools of crowd workers, relaxing the assumption that pool size is fixed and providing support for multiple crowd-powered applications running in parallel on the same pool of workers. I introduce Cioppino, a multi-tenant crowd system inspired by analogies between crowd worker management and cloud infrastructure management. Cioppino is architected around a queuing model of worker behavior, and leverages novel techniques to improve the performance and efficiency of the crowd, including control-theoretic auto-scaling of the crowd in response to changing application workloads, adaptive recruitment to compensate for worker abandonment, and proactive transfer of workers between applications to maximize utilization. I evaluate Cioppino in simulation on a trace derived from workers hired by the CLAMShell system of Chapter 4, and demonstrate that the system's techniques can dramatically decrease cost, while improving performance and respecting workers' preferences for which tasks to work on.

In Chapter 6, I describe my work on the problem of identifying fast workers in an anonymous crowd. The optimizations of Chapters 4 and 5 rely on access to a pool of quick workers, and this work provides theoretical guarantees on how efficiently such a pool can be identified among an unknown population. Finding fast workers can be connected to a statistical setup called the most biased coin problem. I prove lower bounds on the hardness of this problem, importantly demonstrating that strategies that try to learn about the crowd population and then use that knowledge to find fast workers are quadratically more expensive than adaptive algorithms that simultaneously learn and search. Subsequently, I present a novel adaptive algorithm that finds fast crowd workers with no prior knowledge of the worker population distribution parameters, and prove near-optimal guarantees on its runtime. These algorithms have important consequences not only for crowd systems but for any sequential decision process.

Finally, in Chapter 7 I summarize the main contributions of this work and discuss some of its limitations. I conclude by suggesting future opportunities for improving the state of performant crowd systems, and offer a few final thoughts on the broader implications of designing hybrid human-machine software.

Chapter 2

Background

The term "wisdom of the crowd" is frequently invoked to describe the power of bringing the attention of many people to bear on a single difficult problem. However, there is a wide gap between the understanding that crowdsourcing can be an effective problem solving tool and the ability to design software systems that efficiently leverage human expertise. Over the past decade, significant progress has been made in the latter, to which this chapter provides a gentle introduction. In particular, because this thesis focuses on the performance of human-powered systems for data analysis, I focus on background related to data analysis systems.

In addition to describing how state of the art crowd-powered data analysis systems work, I identify key challenges in their design, particularly those related to system performance. Throughout, I describe related work both in the area of human computation systems and in the several other areas from which my thesis draws.

2.1 The Cast of Characters

When reading a thesis about computer systems that assign work to humans, it is easy to get confused about which humans are doing what. There are two main roles played by humans in human computation systems: those who request work and those who perform it. Here, I briefly highlight the terminology used to refer to each, both in this thesis and in other work.

Users, alternately *data analysts* or *requesters*, are the humans who are seeking to use a human-powered system to accomplish a data analysis goal such as training a machine learning model or cleaning a data set. They load datasets into the system, issue high-level commands (e.g., 'clean the `date` column of the `Restaurants` table using the crowd'), and use the results of the analysis after it has completed (e.g., run a query on the cleaned up `Restaurants` table).

Workers[1], alternately *the crowd* or *human processors*, are the humans who actually analyze data, coordinated by the system itself. They are employed internally by companies or hired on platforms such as MTurk, and repeatedly interact with the system's interface to perform tasks such as data labeling or de-duplication. Systems that optimize performance by managing their workforce, such as those described in Chapters 4 and 5, hire and release workers using automated algorithms, but other systems may make hiring decisions in a more traditional way (i.e., through an HR department) or by leaving hiring decisions to the first-come, first-served policy used by most crowd marketplaces today.

Though in some systems, these roles are easy to identify and separate, in others it can be less clear. For example, in some human-in-the-loop data cleaning systems (e.g., [58]), the system user also acts as an expert worker by processing example data items. For this reason, it is preferable to think of users and workers as roles, rather than individuals, understanding that a given person may embody both at once.

2.2 Human Computation Platforms

The workers who process data must be hired from somewhere, and there are numerous platforms that serve as marketplaces to connect users with workers. In this section, I overview some of the platforms most commonly used in human-powered data analysis systems and describe their features. All of them have in common the feature that tasks can be sent to workers programmatically, separating them from traditional employment marketplaces and recruiting tools.

Internal Crowd Platforms. A recent survey found that about 50% of companies that use crowd work host platforms developed in-house to coordinate workers rather than hiring them on public marketplaces [100]. Larger technology companies in particular have received attention for their use of contracted labor on internally-developed platforms for crowdsourcing, including Twitter [12], Facebook [150], and Google (both in their Youtube [4] and Search [47] products). Workers are either full-time employees or (more commonly) contracted from agencies such as ZeroChaos [149] or TaskUs [134] that specialize in outsourcing and temp jobs. Once hired, workers are coordinated using proprietary software for tasks including data categorization, content moderation, and data analysis (see Section 2.3.5 for common specific crowdsourced data analysis applications).

Microtask Crowd Platforms. The most popular public crowdsourcing platforms for data analysis specialize in *microtasks*: small, simple questions with well-defined outputs that require little context to complete, for example image labeling or text translation. By far the most popular such platform is Amazon's Mechanical Turk (MTurk) [105], which connects

[1]Note: the terminology becomes unavoidably more complicated because computers in a distributed cluster are also frequently referred to as workers. In this thesis, I explicitly modify worker with 'machine' or 'node' unless it is clear from context that I am referring to a computer.

users with a workforce of over 500,000 workers from 190 countries. MTurk has been widely adopted by the academic community because of its convenient APIs for creating tasks, hiring workers, and fetching results. Common tasks on the platform include translation, transcription, image processing, and data extraction or collection. Other popular microtask platforms include CrowdFlower [39], which sources workers from multiple partner 'channels' and offers several additional features such as quality control in addition to simply creating tasks, and Samasource [123], which focuses on recruiting workers from disadvantaged populations in the developing world and organizes them via agencies that provide office space and computers to perform microtask work.

Macrotask Crowd Platforms. Some data analysis work, for example complex data extraction [63], cannot be accomplished via short microtasks. To complete such tasks, workers are typically hired as freelancers in longer-term contracts on *macrotask* platforms. Upwork [138] (formerly Elance and Odesk) is the best known macrotask platform. Its freelancing model is much closer to traditional employment than microtask platforms, and in addition to performing crowd-powered data analysis tasks, workers are hired to perform work such as web design, copy editing, and software development. As a result, the platform is heavily focused on connecting users with workers who have the necessary expertise, experience, and availability. However, the site still provides programmatic access to contracting, task assignment, and payment.

Domain-specific Crowd Platforms. The previously described human computation platforms are entirely generic, in the sense that users can design their own interfaces and ask workers to perform any type of work. However, there are also a number of crowd platforms tailored to employing workers for specific types of work, both microtasks and macrotasks. For example, Tamr [132, 130] leverages human workers to assist in data integration microtasks, and LeadGenius [92] (formerly MobileWorks [106]) employs the crowd to manage the process of sales lead generation and management (composed of complex macrotasks).

Platform Features. Beyond allowing programmatic access to a human workforce, most human computation platforms provide additional features to improve the experience of requesters. For example, MTurk and Crowdflower both provide ready-made templates for task interfaces in common categories such as image tagging or data collection so that requesters need not design their tasks from scratch.

One important feature is task quality control, which seeks to minimize errors in human work. Crowdflower provides out-of-the box quality control by automatically sending tasks to multiple workers and inferring the result based on inter-worker agreement. Though other platforms do not offer this feature, their users frequently implement it anyways using custom code. MTurk also allows workers to be filtered based on the quality of their previous tasks in order to improve the likelihood of high quality work. Platforms such as Upwork and

Samasource provide tools for users to spot check workers' output, an important technique for macrotasks that cannot be assigned to multiple workers.

Additionally, all platforms allow users to provide feedback to their workers. MTurk lets users reject tasks that are poor quality with explanations to help the worker improve, and Samasource, Crowdflower, and UpWork allow for detailed feedback around each task as well, though Samasource typically expects managers in its agencies to provide training rather than depending on requesters.

Finally, platforms typically provide mechanisms for assessing the quality of workers. MTurk uses quantitative measurements such as percentage of tasks accepted vs. rejected, allows users to design custom qualification tasks for workers, and has its own 'master' qualification to identify high quality workers. Crowdflower organizes workers into three tiers of quality, allowing workers in the top tiers access to more interesting work with higher pay. Additionally, it uses tasks with known answers to explicitly assess worker quality, and uses some of the algorithmic techniques described in Section 2.4.1 to infer quality from completed work. Upwork also allows for task-specific qualification tests, and uses its public feedback system to track the reputation of its workers.

2.3 Human-in-the-Loop Data Analysis

The high-level goal of a human-in-the-loop system is to divide labor between automated data processing and human worker contributions in a way that takes advantage of the speed of software without losing the insights of the workers. A key property of such systems is *declarativity*: data providers typically specify data analysis tasks at a high level without prescribing the details of the division of labor. This creates opportunities for optimization. In particular, the workflow describing how data is passed between workers and system, the selection and management of the workers employed by the system, the assignment of particular workers to particular data points, and the interface that workers use to operate on data can all be modified while still meeting the requirements of the data provider.

Nevertheless, most human-powered data analysis systems follow a similar overall workflow (illustrated in Figure 2.1), wherein batches of data are repeatedly sent to workers for processing until the desired analysis is judged to be of sufficient quality. The following subsections examine this process in detail in the context of an example application.

2.3.1 Overview and Example Application

Though the specific type of analysis performed by a human-powered system may vary, the flow of data throughout the system is fairly consistent across applications. For ease of exposition, I will describe this flow in the context of a simple example from data cleaning: the goal is to remove duplicate products from a dataset using the crowd, a process known as Entity Resolution. To do so, the system asks crowd workers to compare pairs of products and answer the binary question, "Do these two product descriptions refer to the same real world

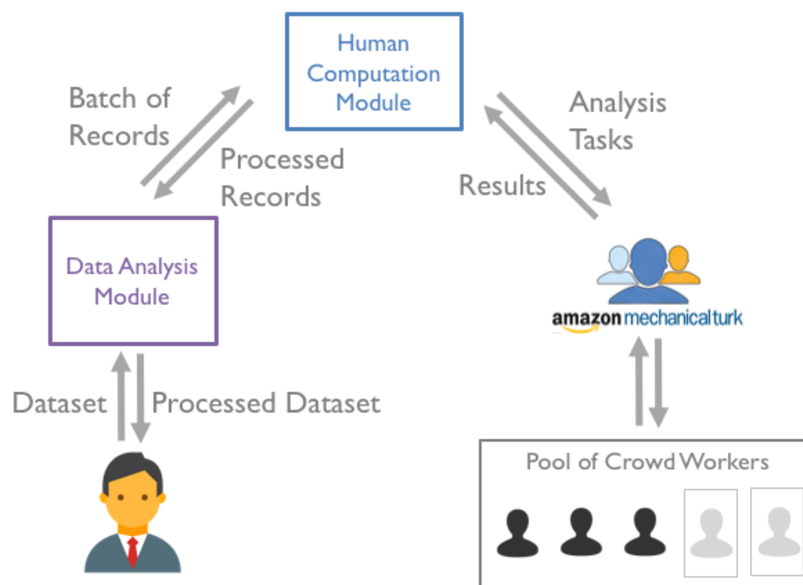


Figure 2.1: The flow of data through a human-powered data analysis system.

entity?" Because product databases can be very large and pairwise comparison is quadratic in the size of the data, it may be impractical to ask workers to compare all pairs of products in the dataset. Instead, typical entity resolution systems acquire enough answers from the crowd to train a binary classifier that can identify a pair of products as duplicate or distinct, then use that classifier to label the rest of the data.

Now let us consider a single execution of our entity resolution application on a dataset of products, following the flow in Figure 2.1. First, the Data Analysis Module selects a batch of product pairs to be labeled, and sends it to the Human Computation Module. The Human Computation Module synthesizes each pair into an interface that can be presented to a worker, for example, an HTML form with radio buttons for 'duplicate' and 'not duplicate'. It then uses the API of a platform such as MTurk to publish the interface as a task. Once the task is published, a worker is assigned to complete it. When the worker completes the task, the HTML form is submitted, and the worker receives payment for the work. The submitted responses are aggregated by the Human Computation Module, and when the whole batch is complete, the processed batch is returned to the Data Analysis Module. The Data Analysis Module uses the labeled batch to train a binary classifier, and evaluates the quality of the classifier against held-out data using a technique such as k-fold cross-validation. If the classifier is judged to be of sufficient quality, it is used to assign labels to all remaining pairs of products, and the analysis is complete. If not, the Data Analysis Module selects another

batch of data to be processed, and so forth until an accurate classifier has been trained.

2.3.2 Selecting Records to Process

How does the Data Analysis Module pick which records to put into each batch? Picking a small subset of records to be representative of the entire dataset is essentially a population sampling problem, so any number of techniques from the statistical literature might be appropriate, but there are several common approaches:

Random sampling. The simplest and most common approach to batch selection is to simply pick a set of records uniformly at random from the data set. This can be desirable, for example, when fitting a supervised learning model that assumes that its training data are independently distributed. For applications where machine learning is not involved, random sampling is a reasonable default if there is no domain knowledge to inform the choice.

Adaptive sampling. Batch selection can also be accomplished using active learning techniques (see [126] for a survey) that leverage information about previously-processed data to pick the next batch. Intuitively, this approach makes sense: if, for example, the system has already identified most of the duplicate smartphone records in the product database, it is likely more useful for the classifier to observe examples of duplicates among other products than to be given another example of two matching smartphones. When employed appropriately, active learning can dramatically reduce the number of records the crowd needs to process before a high-quality classifier is obtained. As a result, a number of recent systems have leveraged active learning to select batches of data to send to the crowd [58, 104].

Stratified sampling. When information is known about natural groupings in the dataset, records can be selected based on those groupings. For example, if the product database contains a column describing the category of each product, it clearly makes sense to prefer record pairs within the same category (e.g., two smartphones) over pairs that span very different categories (e.g., a smartphone and a soccer ball). In systems for entity resolution, these domain-specific criteria for picking records are called *blocking rules*, and are employed universally [142, 58]. Stratified sampling is also important for applications performing approximate query processing with the crowd [70], particularly when records are unevenly distributed among groups. The SampleClean system takes this into account [143].

2.3.3 Synthesizing Task Interfaces

Once a batch of records has been selected, each must be presented to workers in an interface that allows for efficient and accurate data processing. Because most human computation occurs via a web interface, interfaces are almost exclusively described in HTML. The standard approach to generating these interfaces is template-based: each task type is assigned an

HTML string that is complete except for data-dependent fields, represented by placeholders. To synthesize the interface for specific data records, these fields are extracted from the records and substituted for the template's placeholders. Templates for specific task types can either be hardcoded in the system (the default approach), generated based on the semantic operation performed (e.g. 'sort' or 'check equality') [52], or registered via system APIs [102, 63].

2.3.4 Recruiting Workers

The assumption that a worker can be hired on-demand when data is ready for analysis underlies many human-powered data analysis systems. Interestingly, the model of recruitment popularized by MTurk is somewhat inverted from this: available tasks are published in a large listing, and workers choose which tasks to work on according to their preference. This means that a task is not guaranteed to have a worker assigned to it unless the task is sufficiently desirable (i.e., interesting, high-paying, etc.). The mismatch between system assumptions and platform realities has significant consequences for the performance of crowd-powered systems, which has led to the development of several alternate recruitment models, including:

In-house crowds. Especially at larger companies, platforms like MTurk are increasingly eschewed in favor of directly hiring contract or temp workers in-house to perform manual data analysis tasks [4]. These workers are paid hourly or proportionally to the number of tasks completed. This approach gives companies much more control over the size, training, and skill of the available workforce and the speed with which new tasks are assigned to workers.

Retainer pools. First suggested by Bernstein et al. [16], the retainer model builds on top of existing platforms to explicitly place task assignment in the hands of the system. Retainer pools recruit workers in advance of work becoming available and pay them cents per minute to wait if there is no work available (though they are free to perform other work during that time). When the system has new tasks to process, it directly assigns them to workers waiting in the pool. The retainer model has been shown to effectively eliminate the overheads of recruitment time, and the added cost of paying workers to wait is not unreasonable when amortized over a large batch of data or deployed in a high-load streaming setting.

In this thesis, I focus exclusively on the retainer pool model of worker recruitment. This is because, as I demonstrate in Chapter 4, not doing so has severe consequences for system performance. The design of performant crowd systems cannot be accomplished on top of a worker recruitment model that undermines key assumptions about worker availability.

2.3.5 A Sample of Applications

The entity resolution application I have used as an example thus far, while illustrative, is only one of the many uses of crowd-powered data analysis systems. The workflow described in Figure 2.1 is quite general, and a wide variety of applications have been developed using it. Here I describe some of the most common classes of human-in-the-loop data analysis applications.

Machine learning. Due to the recent surge in the volumes of data collected worldwide and the ease of acquiring the computational resources to process them, there has been an increasing ubiquity of use cases for machine learning. The most widely used machine learning techniques fall into the category of *supervised learning*, in which a predictive model is built by running a statistical algorithm on a labeled training set. In the rare instance that users already possess labeled data, this works well. However, for many use cases, a labeled training set does not exist a priori, and the labeling process is often complex enough that human input is needed to generate accurate labels. As a result, crowd-powered data labeling for machine learning has been widely adopted in industry [100], and the academic community has designed approaches for generating training data [46, 91], using active learning techniques to pick the most useful training points [58, 104], or using crowd workers to generate useful features for machine learning [33]. Because of humans' natural expertise, many crowd machine learning applications focus on the areas of vision and natural language understanding.

Data cleaning. Another consequence of the availability of large quantities of data is the opportunity to analyze data combined across many different datasets. Unfortunately, merging datasets frequently leads to missing, incorrect, or duplicate data. Though some errors can be fixed only by domain experts, others are amenable to processing by non-expert crowd workers, and crowd systems have been proposed for tasks such as integrating multiple datasets [130], performing entity resolution to eliminate duplicates [58, 142], fixing missing or incorrect values [103, 112], and supporting end-to-end multi-stage cleaning workflows [34].

Data extraction. Much of the data on the web is locked up in hard-to-analyze formats such as HTML, pdfs, flash animations, or even paper forms. Data extraction techniques seek to identify the structure in such data and translate them into more machine-readable formats, often relational tables. Again, though automated approaches to extraction can be successful if the data have relatively consistent latent structure [22, 14], in many cases humans are required to reliably identify structured data. Human-in-the-loop systems have been proposed to extract data from spreadsheets [32], paper forms [30], various web formats [63], and unstructured fields within a structured dataset [82].

Data acquisition. Traditional data systems have been built around the *closed world assumption*: if a record is not in the system, then it does not exist. In reality, this assumption

seldom holds, as there are no existing systems that capture universal knowledge. Crowdsourcing offers a unique opportunity to relax the closed world assumption, since workers can be asked to search for and contribute knowledge to a system storing incomplete information. As a result, crowd-powered systems have been proposed to answer open-world relational queries [52, 102, 110], enumerate lists of unknown length [137], fill in tables containing unknown data [112], or search for data with properties that can only be verified by humans [41, 109].

2.4 Challenges in Human-Powered System Design

Designers of performant human computation systems must resolve a key tension between human behavior and programmatic abstraction. On the one hand, we would like to build high-quality, easy-to-program, low-latency, low-cost, high throughput systems that treat workers as abstract computational resources. On the other, we must acknowledge that workers are not CPUs, and that there are human factors that will confound performance if we don't take them into account. As a result, a good system abstraction must model enough of these factors to avoid surprises in performance without falling down the rabbit-hole of attempting to capture all of human behavior. In this section, I discuss the challenges that arise when using human workers to power systems and how the existing literature has addressed them.

2.4.1 Quality Control

One important difference between workers and computers is that workers have varying levels of skill and experience, and may not complete work with consistent accuracy over time due to familiarization or fatigue [74]. As a result, crowd-powered systems cannot assume that the output of a task is guaranteed to be 100% accurate, and a prominent focus of the crowdsourcing literature has been on *quality control*: mitigating errors introduced by workers. Past errors made by workers provide an important signal for predicting errors made on the current task, so many approaches to improving task quality involve simultaneously fixing errors and learning worker quality.

Most of the tasks necessary for data analysis (with the notable exception of data extraction, see, e.g., [63]) fall into the category of *microtasks*: short, simple work that requires little to no training and has a well-defined categorical output. For example, entity resolution task outputs are binary yes/no answers, and image object detection task outputs are categorical.

Quality control for microtasks has received the most attention in the literature. One approach to quality control is algorithmic. These techniques weigh worker responses based on quality inferred from previous answers [75, 83]. Beyond simple weighted averaging, other algorithms have been adopted from the statistics literature to address crowd quality, most popularly Expectation Maximization [42] algorithms that simultaneously estimate the answers to tasks and the error rates of workers [83, 80]. Another approach to quality control

uses additional crowd workers to improve the quality of tasks. For example, the Find-Fix-Verify design pattern [18] verifies task outputs by having other workers vote on their correctness. Finally, machine learning can be used to predict and control quality. Rzeszotarski et al. [122] use behavioral measures such as mouse movements and task completion time to train a predictive model of task error.

In complex *macrotasks* where task output may not be binary or categorical, quality control is more challenging. The main approaches include using machine learning based on task-specific and worker-specific features to predict task quality [63] and organizing workers into hierarchies so that more senior workers can efficiently identify and correct errors in work completed by those less experienced [63, 106].

In practice, most real-world crowd deployments use a combination of two simple approaches. First, so-called "gold standard tasks" with known answers are assigned to workers and used to assess their baseline accuracy, and workers with low quality are excluded (CrowdFlower [39] provides this service, for example). Second, incoming tasks are assigned to multiple workers, and a majority vote of their responses is used as the task output. These techniques, though repeatedly shown to be less effective then the more sophisticated approaches described above, are easy to implement and frequently perform "well enough" for many use cases.

2.4.2 Cost

Though there exist platforms for 'volunteer-sourcing' data analysis (CrowdCrafting [38] and GalaxyZoo [54], for example), workers employed to process data typically expect to be paid. As a result, systems making heavy use of human computation need to account for the monetary costs associated with crowd labor. Workers are typically paid in one of two ways: on a per-task basis, with each completed task earning a few cents, or hourly, based on a fixed rate. The retainer model described in Section 2.3.4 does both at once: idle workers are paid an hourly wage to wait for work to arrive, and active workers are paid for each task they complete. These costs can grow quite large, especially as the size of the dataset being processed grows. There are two main classes of technique used by crowd-powered systems to reduce costs. One class reduces the fraction of a dataset examined by the crowd to lower the number of tasks that must be paid for. The other class dynamically adjusts task features like pricing or redundancy to pay the minimum amount necessary to meet system performance and quality goals.

Reducing the data processed. Several approaches to reducing cost by considering less data have been successful. Some approaches improve the efficiency of specific operations. For example, in entity resolution, blocking rules eliminate crowd tasks that would compare pairs of items very unlikely to be duplicates [142]. Marcus et al. [99] ask workers to estimate counts (for example, of the number of males vs. females in a large set of images) in order to avoid having a separate task for each data point. Das Sarma et al. [41] find items possessing human-verifiable properties in a dataset using algorithms that can ask fewer questions of the

crowd in order to be cost-optimal. Parameswaran et al. [111] describe multiple workflows for crowdsourced information retrieval that vary the number of questions asked to crowd workers in order to maximize precision and recall.

Domain-independent techniques can reduce costs by carefully sampling the dataset. For example, the SampleClean system uses the crowd to clean a fraction of the data, then uses statistical techniques to estimate query results on the whole dataset [143]. Additionally, machine learning techniques show the crowd a small set of training examples, then extrapolate from that data to process the remainder of the data set. Active learning techniques in particular, which iteratively use a point selection algorithm to pick a small set of informative points to label, then retrain their model on the new labels, are useful for reducing the amount of training data needed to produce an accurate model [36, 126]. As a result, they have been used in several crowd-powered systems to reduce cost [58, 104], and I make use of them in Chapter 4.

Adjusting tasks for efficiency. In addition to simply considering less data, crowd-powered systems can reduce costs by modifying the way in which tasks are run. For example, interfaces can be redesigned to make workers more efficient. A common strategy is batching, where a worker processes multiple data records in a single task [99, 101, 142]. In a batched interface, a worker is paid slightly more per task, but the system saves money per record because each task contains several records (5-10 records per task is typical). Another way to save costs is to dynamically adjust pricing in order to pay workers just enough to encourage them to accept and process tasks fast enough to meet system performance or quality goals [55, 24]. Finally, in settings where multiple workers process each datapoint in order to improve quality, cost can be reduced by adaptively varying the number of workers assigned to a record based on the expected quality improvement of getting another opinion [127].

2.4.3 Performance

Perhaps the most obvious drawback to embedding humans in automated data analysis systems is the fundamental difference in speed between a human and a computer: computers perform billions of operations per second, but even at their fastest, humans can only process one or two records in a second. In addition, human response times tend to exhibit high variance, as a multitude of factors from inherent speed to temporary attention or fatigue differ between individuals. In order to prevent human steps from becoming the bottleneck in automated data analysis, crowd-powered systems must find ways to mitigate the latency and variance of worker response times. Crowd system performance has only recently received attention, as it is significantly harder to reason about and experiment with than cost or quality. Here, I list some important considerations that should be taken into account when designing crowd systems for performance, and survey related work that has addressed them.

Problem domain. The easier the crowd work, the less potential for human factors to affect system performance. For example, microtasks such as image labeling where the result is objectively correct or incorrect, the output data is well structured, and the data domain is easily understood by most workers can be incorporated into systems that use simple API calls to show workers tasks and aggregate their results. Systems that intentionally decompose complex problems into microtasks [18, 95, 85] or those that recruit experts who better understand the problem domain [119, 90] explicitly address this issue.

Desired latency and throughput. When workers are processing tasks at high speed, they will not behave as consistently as CPUs. It is well-known that crowd workers take breaks, exhibit fatigue or lack of attention to tasks, and eventually abandon the system entirely [121, 17]. If latency is an important system metric and individual workers complete many tasks, these human factors must be taken into account. For example, inserting micro-breaks [121] can keep workers better focused during actual working time, and careful task design can help reduce the latency of the crowd [86, 99]. In Chapter 4, I describe techniques to reduce latencies by an additional order of magnitude, and in Chapter 5, I discuss how to improve throughput when supporting multiple low-latency crowd systems simultaneously.

Desired availability of crowd workers. Workers cannot be reliably provisioned on demand like cloud compute resources, and exhibit incredibly high variance in the latency with which they accept tasks posted on marketplaces like MTurk. In addition, for complex work, new workers may require overhead for onboarding and training. Systems where availability is a concern use techniques like retainer pools [16], variable task pricing [55, 24], and over-recruitment [20] to ensure that recruitment time doesn't become an issue.

2.4.4 A Focus on Latency

This thesis describes novel techniques and algorithms for improving the performance of crowd-powered data analysis systems. Though the previous section provided an introduction to the performance challenges of crowd systems, properly motivating latency reduction in crowd systems requires a deep understanding of where latency in human systems arises. This subsection comprehensively lays out the potential sources of delay in a crowd system to provide that understanding. Figure 2.2 shows the sources of latency in the context of the crowd-powered system workflow introduced in Section 2.3.

A multitude of factors can increase latency, from algorithm choice to worker and environmental factors. Categorizing the factors based on the granularity of work provides a clear decoupling of algorithmic contributions from systems concerns. Specifically, latency might arise from the speed of a single task, a fixed batch of tasks, or the full run of multiple batches (of possibly varying sizes).

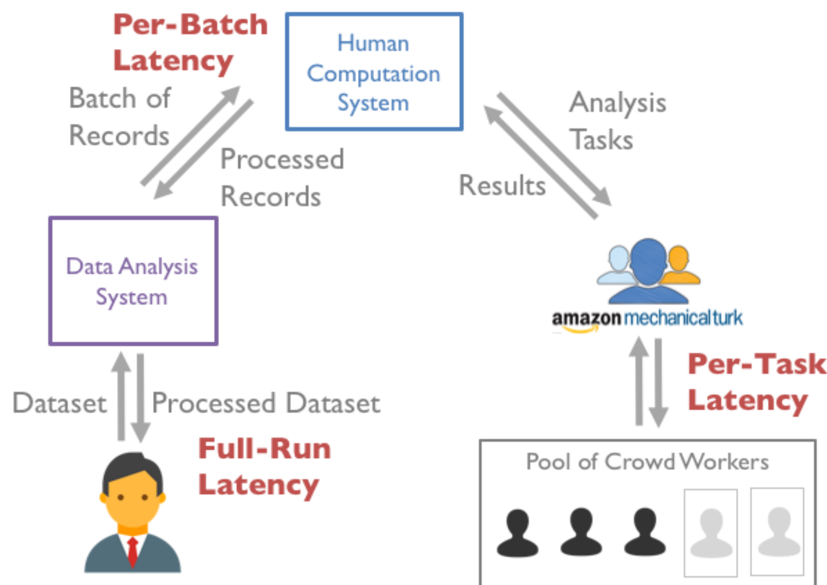


Figure 2.2: The flow of data through a human-powered data analysis system shown in Figure 2.1, with sources of latency labeled.

Terminology. To better characterize the flow of tasks in a crowd system, I first introduce some terminology. A *task* is a single unit of work[2] that is assigned to a worker. The *per-task* latency is the time between the task's creation on the crowd platform (via an API call such as the `CreateHIT` call of Table 1.1) and the worker's submission of the completed work. Multiple tasks are grouped into a *batch*, which is sent to the crowd and processed by multiple workers in parallel. The *per-batch* latency is the time between the creation of the first task in the batch and the submission of the last task in the batch. The next batch does not begin until the current batch ends. Tasks in each batch are sent to a *retainer pool* of crowd workers (which I will refer to interchangeably as the *pool*, the *crowd pool*, or the *worker pool*). Workers in the pool are assigned tasks in parallel, and receive a new task as soon as they submit a completed one. An *application* is data analysis software that runs multiple batches of tasks serially. An application's *full-run latency* is the time between the start of an application's first batch and the end of its last batch.

Per-task latency. The latency of a single task can be viewed as arising from a linear sequence of three phases:

[2]though it may contain multiple data records, e.g. an entity resolution task that asks a worker to compare two items.

1. Recruitment: workers do not immediately begin working on newly submitted tasks, and recruitment latency consists of the time until an interested crowd worker accepts a newly posted task. Due to the worker-oriented task assignment policies of platforms like MTurk, recruitment times of minutes to hours are not unusual.

2. Qualification and Training: once workers accept a task for the first time, they are often presented with tutorials or qualification tasks before they are permitted to perform actual work.

3. Work: the amount of time a worker spends to complete a task can vary depending on the worker competency, the time of day, fatigue, and numerous other factors [89, 74]. Note that a single task may produce multiple labels if records are batched into tasks (a common practice).

Per-batch latency. The latency of a fixed-size batch of tasks is the time for all tasks in the batch to receive responses. Batch latency is affected by the *latency distribution* of all available workers, in addition to each worker's individual variations. On microtask platforms like MTurk, it is common to observe extremely slow batch completion times. This effect is driven by three sources:

1. Mean Pool Latency (MPL): If all workers processing a batch are slow on average, the batch latency will be high. Different workers have different average processing speeds, so identifying fast workers is important.

2. Pool Variance: because the batch cannot complete until every task receives a response, per-batch latency is set by the slowest workers in the pool. In a high variance pool, the slowest workers are significantly slower than the average worker. Workers who take breaks or are unusually slow can exhibit work times of up to 3 orders of magnitude slower than the median, significantly increasing the per-batch latency.

3. Worker Variance: individual workers do not complete every task in the same amount of time. At the high end, workers have been observed with standard deviations of over 2 hours. In order to program reliable crowd-powered systems, this variance must be accounted for.

Both worker and pool variance increase the frequency of *straggler tasks* that are abnormally slow compared to the rest of the batch. Reducing the MPL and mitigating the effects of stragglers are therefore important for keeping the per-batch latency low.

Full-run latency. Rather than require crowd workers to label terabytes of data, machine learning is often used to infer labels once enough records have been labeled to train a high-quality model. Active learning can reduce the size of this training set, however training the model requires acquiring small batches of labels in a blocking fashion. This induces four latency sources:

1. Decision Latency: The time to pick the next batch of tasks (e.g., uncertainty sampling for active learning).
2. Task Count: The number of labeling tasks, which machine learning approaches seek to reduce.
3. Batch Size: The batch size affects both active learning convergence as well as the amount of parallelism within a batch.
4. Pool Size: The number of workers completing tasks controls the maximum parallelism possible, however is often dictated by operational constraints.

Active learning can drastically reduce the task count, but incurs increased decision latency and requires limited batch sizes to be effective. In contrast, passive learning (selecting random records in each batch) can leverage the parallelism of all available workers, but might require many more tasks to train a model of equivalent accuracy. The choice ultimately depends on the task.

The variety of potential sources of delay underscores the fact that crowd-powered systems are not slow just because people are slow. Rather, the realities of passing data between human and automated components and effects related to trends in the overall population of workers contribute significantly to latency, and unlike human nature, these effects can be identified and corrected. In this thesis, particularly Chapter 4, I consider and account for each of the sources of latency above when designing performant crowd-powered systems.

2.5 Other Related Work

One of the main contributions of this thesis is the insight that if a crowd of workers can be modeled as a distributed cluster of computers, well-known methods for programming clusters efficiently can inform how we approach programming humans. As a result, I draw on a wide range of knowledge from other fields, ranging from the theory of infinite-armed bandits to techniques for auto-scaling cloud software services. Though each of these fields deserves (and has received!) significant attention in its own right, I use this section to introduce the ideas most relevant to my work. Subsequent chapters will dive deeper into the literature specifically relevant to that chapter.

2.5.1 Declarative Systems and Database Optimization

The use of relational databases to maintain and query organized collections of data dates back to the 1970s [35]. Key principles introduced by the field include data independence, which states that the way applications refer to data need not correspond to the way that the system stores the data, and declarativity, which states that queries should ask for *what* data

they want, *not how* the data should be retrieved (The SQL language [8] is the most well-known instance of declarative querying). Together, these two principals allow for powerful optimizations [125], since the system can change how data is stored and retrieved without affecting how the user refers to and asks for data.

Data independence, declarativity, and optimization are all key principles underlying crowd-powered data analysis systems. The first crowd systems for data analytics were designed as relational databases [52, 102, 110] that extended the SQL language with crowd primitives and operators, allowing for the existing optimizer to plan queries that required human involvement. Further, all of the performance optimizations presented in this work are made possible due to a strict declarative interface. The user provides a dataset and an analysis task, but because of the independence between this API and the details of assigning specific records to specific workers, the system is free to change assignment schemes, manage workers, and adjust redundancy.

2.5.2 Distributed Cluster Computation

With the rise of modern datacenters and the introduction of the MapReduce paradigm [43], it is increasingly common to run programs on large datasets by distributing the computation across a cluster of coordinated computers, using frameworks such as Hadoop [68] or Apache Spark [147]. These frameworks typically operate according to the Bulk-Synchronous Processing (BSP) model, wherein computation proceeds in rounds alternating between independent computation by all machines and synchronization barriers where results are shared between machines [139]. Because of the desire to shield framework users from the complexities of maintaining a cluster, scheduling computation on worker machines, and sending computation and data across the network, these frameworks have developed techniques to automatically handle failed compute nodes [43, 147], schedule which tasks run on which machines [44, 140, 108], share hardware resources between multiple users applications running on the same cluster [56], and mitigate the effects of slow, or *straggler* machines [43, 10, 11, 148].

A key analogy in this thesis is that a crowd of workers can be thought of as a cluster of machines coordinated by a crowd-powered system, and many of the challenges of distributed computation have analogues in the crowd setting. For example, the workflow in Figure 2.2 is an instance of BSP, with synchronization barriers at the end of each batch of tasks. As a result, techniques for accelerating BSP computation (straggler mitigation [10], for example) can be used to reduce the crowd system's per-batch latency. I apply this technique in Chapter 4. Similarly, crowd systems must react to workers abandoning the system mid-task (like a machine failure in a cluster), and assign tasks to workers based on their preferences, skills, and speed (like task scheduling in a cluster). Additionally, crowd systems that support multiple user applications must share crowd workers efficiently but fairly between applications, like a multi-tenant cluster. I describe performant approaches to handling worker abandonment, task assignment, and worker sharing in Chapter 5.

2.5.3 Cloud Infrastructure Management

Because the number of users of modern web applications can increase or decrease rapidly over short time periods, a common architecture used to ensure consistent performance for users is to run the application on multiple stateless worker machines, with a separate load balancer machine that directs incoming requests to workers with available resources. However, picking the right number of worker machines to handle the number of incoming requests is challenging, especially if workloads change over time. As a result, both industry cloud providers [6, 59] and researchers [107, 79, 5] have focused on *autoscaling*: adaptively increasing and decreasing the number of worker machines to support changing workloads. [116] provides a recent survey of techniques for autoscaling, which include using static rules, control theory, or machine learning to adjust the cluster size.

Again, crowd systems face a similar problem: how many workers should be hired to keep up with incoming requests for data analysis tasks? This issue is particularly important under the retainer model, where worker hiring decisions are fine-grained and under complete control of the system (as opposed to the standard model where workers choose which tasks to work on, effectively assigning themselves to the system workforce). I describe techniques for autoscaling pools of crowd workers in Chapter 5.

2.5.4 Multi-Armed Bandits

The performance of a crowd-powered system is limited by the speed of its workers, so when making hiring decisions, it is important for the system to take speed into account. However, on most crowd platforms, hiring via the API allows arbitrary workers from the global population to pick up a task, so the system doesn't know *a priori* any information about either how fast a new worker is, or what the distribution of speeds in the population is like. It therefore must navigate a delicate tradeoff between learning more about the workers it has already hired or taking a chance on a new worker in the hope that they will be fast compared to the existing workforce. This tradeoff can be seen as an instance of the multi-armed bandit problem (see [77] for a brief survey), which provides a useful framework to reason about crowd hiring.

The multi-armed bandit problem[3] proceeds in a series of rounds. In each round, the player chooses an 'arm' i from an available set of K arms, and observes information about that arm (typically given as a sample from an iid random variable with mean μ_i), with the goal of identifying the arm associated with the highest mean μ_* using as few pulls as possible. If we think of the player as a crowd-powered system, the arms as all workers in the population (already hired if the arm has been chosen, unknown otherwise), and the 'choice' of arm i as sending a task to worker i and observing the worker's response time, then the game describes algorithms that attempt to quickly identify workers with high average speeds. Because the total population of workers who might be hired on a platform like MTurk is

[3]The problem is named for and inspired by the challenge of choosing from a casino full of slot machines (one-armed bandits), each with different probabilities of hitting the jackpot.

potentially very large, a useful modification of the problem setting, known as *infinite* armed bandits [25], allows the player to hire potentially infinitely-many new workers, each of whom get assigned μ_i according to a population distribution that describes the overall speeds of all workers. Work in this setting has focused on an assumed population distribution that satisfies $E\epsilon^\beta \leq \mathbb{P}(\mu_i \geq \mu_* - \epsilon) \leq E'\epsilon^\beta$ for some $\mu_* \in [0,1]$ where E, E' are constants, ϵ tends towards zero, and β is known [19, 145, 21, 25]. This constraint bounds the tail of the distribution, preventing scenarios where arms with mean μ_* are very rare compared to the rest of the distribution. However, the parameter β is unlikely to be available for the population of workers on a crowd platform, making the approach impractical.

In Chapter 6, I consider the infinite armed bandit problem with a different population distribution that is amenable to analysis even when the player has no knowledge about the population. I prove lower bounds on the number of task assignments necessary to find fast workers, and develop novel near-optimal algorithms for doing so, still without knowing the parameters of the population distribution.

Chapter 3

Wisteria: An Interactive Data Cleaning System

Having given some context and introduction to the design of crowd-powered systems for data analytics, I next examine in depth one particular application that requires crowd-powered data processing at interactive latencies. Though several examples of crowd-powered systems were described in the previous chapter, it was not necessarily obvious that they needed to have low-latency human input. For example, machine learning models are often trained offline, so it could be argued that there are no time constraints on using the crowd to label training data.[1] Data cleaning, on the other hand, is a challenge not only hard enough to require human input, but integrated tightly enough with subsequent analysis steps that if it runs slowly, it will delay the entire data analysis process. In this chapter, I describe Wisteria, a system that allows users to design multi-stage data cleaning workflows (referred to as *cleaning plans*) with human components that can be modified over time. Because users typically want to make rapid, iterative changes to data cleaning plans, it is imperative that the entire plan, including crowd-powered steps, run quickly enough to provide rapid feedback to the user. The Wisteria system exemplifies the kind of application that relies upon low-latency crowds, and therefore serves as a useful motivation for the techniques described in later chapters.

3.1 Introduction

The ease of acquiring and merging many large-scale data sources has led to a prevalence of dirty data. Unfortunately, blindly using results that are derived from dirty data can lead to hidden yet significant errors in modern data-driven applications, so data must be cleaned before it is used. But because data cleaning is often specific to the domain, dataset, and eventual analysis, analysts report spending upwards of 80% of their time on problems in data

[1] Though as machine learning systems become increasingly interactive and adaptive, this is rapidly changing. See, e.g., [37].

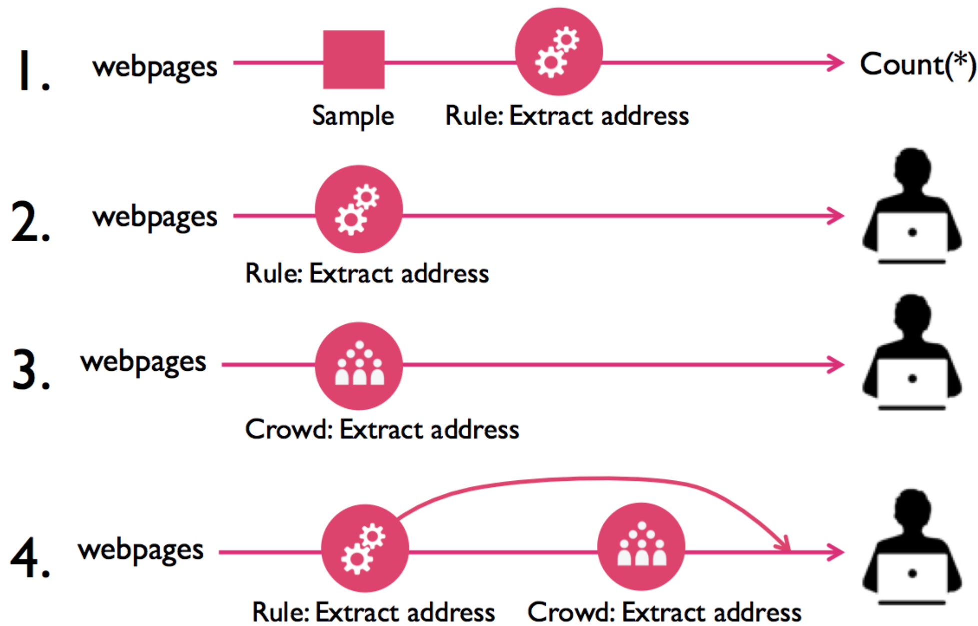


Figure 3.1: Example iterations on the design of the portion of a cleaning plan that extracts restaurant addresses from their unstructured webpages. 1) An exploratory plan that uses a sample to evaluate a simple address extraction method. 2) A plan that applies the method to the entire dataset. The quality is unsatisfactory. 3) An alternate plan that uses manual crowd extraction. The quality is now high, but the crowd-based extractor is slow. 4) A hybrid plan that sends only difficult webpages to the crowd, maximizing accuracy without sacrificing latency.

cleaning [81]. The analyst is faced with a breadth of possible errors that are manifest in the data and a variety of options to resolve them. She must go through the cleaning process via trial and error, deciding for each of her data sources what to extract, how to clean it, and whether that cleaning will significantly change results.

Data cleaning is inherently iterative and Figure 3.1 shows a common progression for the development of a data cleaning plan, in this case the extraction of a restaurant's address from its unstructured webpage. While this operation can easily be represented at a *logical* level by its input and output schema, there is a huge space of possible *physical* implementations of the logical operators. For example, extraction could depend on manually specified rules (*rule-based*), use models trained on previously extracted ground truth records (*learning-based*), ask crowd workers to extract the desired data fields (*crowd-based*), or some combination of the three (e.g., active learning, which uses crowd workers to provide labels for a learning-based approach). Even after selecting (say) a crowd-based operator, many parameters might influence the quality of the output data or the speed and cost of cleaning: the number of crowd workers who vote on the extraction for a given webpage, the amount each worker is paid, etc. *A priori*, a data analyst has little intuition for what physical plan will be optimal

in this large space.

Note that in the evolution of the data cleaning plan in Figure 3.1, our data analyst needed to make many decisions manually about the choice of physical operators by reasoning about their latency, accuracy, and cost. Making the wrong decision, for example using the crowd when it only marginally improves accuracy, can be very costly. A general, scalable, and interactive system that supports rapid iteration on candidate plans would greatly aid this process.

Existing systems seldom address the end-to-end iterative data cleaning process described above. Extract-transform-load (ETL) systems [72, 131, 13] require developers to manually write data cleaning rules and execute them as long batch jobs, and constraint-driven tools allow analysts to define "data quality rules" and automatically propose corrections to maximally satisfy these rules [40]. Unfortunately, neither provide the opportunity for iteration or user feedback, inhibiting the user's ability to rapidly prototype different data cleaning solutions. Projects such as Wrangler [82, 136] and OpenRefine [141] support iteration with spreadsheet-style interfaces that enable the user to compose data cleaning sequences by directly manipulating a sample of the data and applying these sequences to the full dataset. However, they are limited to specific cleaning tasks such as simple text transformations, do not support crowd-based processing at scale, and cannot incorporate user feedback to optimize the physical implementation of the data cleaning sequences. Crowd-based [58, 130] systems have been proposed to relieve the data analyst of the burden of rule specification or manual cleaning, but are usually specific to a single cleaning task (e.g., [58, 112, 103, 32]), preventing end-to-end optimization of the entire cleaning plan. These existing limitations suggest the need for a system that is general enough to adapt to a wide range of data cleaning applications, scales to large datasets, and natively supports fast-feedback interactions to enable rapid data cleaning iteration.

In this chapter, I introduce Wisteria, a system designed to support the iterative development and optimization of data cleaning plans end to end. Wisteria allows users to specify declarative data cleaning plans composed of rule-based, learning-based, or crowd-based operators, then iterate rapidly on plans with cost-aware recommendations for improving the accuracy or latency of a plan. The effects of a plan can be viewed early using sampling and approximate query processing techniques [143].

Supporting these capabilities requires a combination of careful engineering and interface design as well as tackling several research challenges:

- **Sampling**: Wisteria provides sampling as a first-class logical operator for data cleaning plans that tolerate approximation, and uses it to speed up iteration on early-stage plans.

- **Recommendation**: Wisteria recommends cost-aware changes to in-flight cleaning plans that allow users to trade off accuracy and latency, and provides efficient mechanisms for implementing recommended changes without re-executing the plan on already cleaned tuples.

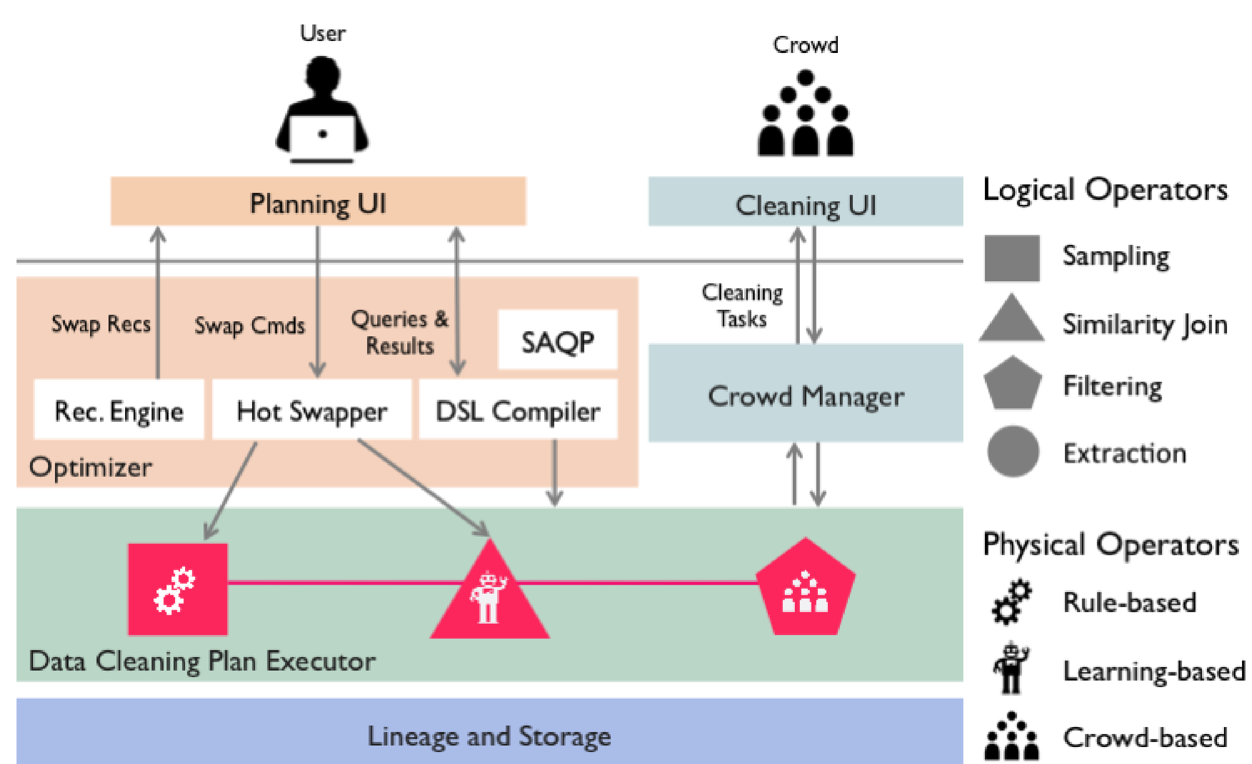


Figure 3.2: Wisteria system architecture, with an example entity resolution plan.

- **Crowd Latency**: Wisteria leverages techniques for straggler mitigation [140] and models crowd worker speed and accuracy to reduce the (often rate-limiting) latency of crowd data cleaning, consistently retrieving results in seconds rather than hours.

3.2 System Architecture

In this section, I provide a brief overview of the Wisteria system and its APIs. Figure 3.2 depicts the system architecture.

3.2.1 Architecture Overview

The Wisteria architecture provides UI, language, and systems tools for building data cleaning plans. Users interact with the system through the **Planning UI**, which allows them to compose data cleaning workflows from modular operators. These workflows are represented as expressions in a novel data cleaning language (section 3.2.2), then synthesized as data cleaning plans by a **DSL compiler**. As the **Data Cleaning Plan Executor** executes the compiled plans, users can interact with the plans via tight feedback loops in two ways. First, users can issue queries to the Sampling-based Approximate Query Processing **(SAQP)**

module and observe approximate results based on the data that has been cleaned thus far. Second, the **Recommendation Engine** displays a set of suggested modifications to the active cleaning plan (for example, making a similarity join more permissive) in the **Planning UI**, and users can update the data cleaning plan in-flight by accepting a suggestion and using the **Hot Swapper** to modify components of the pipeline. Intermediate results and cleaned data are maintained in a **Lineage and Storage engine** that tracks each tuple's lineage in order to enforce the semantics of hot-swapping correctly on in-flight tuples. Logical cleaning operators may have a number of physical implementations (section 3.2.3). Automated rule-based or learning-based operators leverage Spark and MLLib for efficient distributed computation, and operators that require human intervention call out to Wisteria's **Crowd Manager** API, which renders and displays data cleaning tasks to crowd workers from multiple crowds (e.g., Amazon Mechanical Turk) in a web-based **Cleaning UI** for processing.

3.2.2 Cleaning DSL

Wisteria provides a language for specifying the composition of data cleaning operators. The logical operators define the input and output behavior of the operation and the physical operators specify the implementation. The general syntax of this language is:

```
<logical operator> on <relations>
  with <physical operators> , <params>
```

These expressions are composable. For example, the following represents the cleaning plan in Figure 3.2 (an entity resolution plan):

```
Filtering on (
    SimilartyJoin on (
        Sampling on BaseTable
        with Uniform)
    with Jaccard, thresh=0.8)
with CrowdDeduplication, numVotes=3
```

Additionally, Wisteria provides integration of the DSL with Scala/Apache Spark, allowing DataFrames (Spark RDDs with additional schema information) to serve as base tables in expressions.

3.2.3 Cleaning Operators

Wisteria supports a small set of operators that can express a wide variety of common data cleaning workflows. For example, the pipeline depicted in Figure 3.2 performs crowd-based entity resolution: the SimilarityJoin operator generates candidate tuple pairs (the *blocking* step), and the crowd-based Filter operator uses humans to identify duplicates from the candidates (the *matching* step). Additional operators include Extraction and Sampling.

Individual logical operators have multiple physical implementations, each with its own cost, latency, and accuracy profile. For example, crowd-based implementations tend to be high cost, high latency, and high accuracy, whereas rule-based implementations tend to be

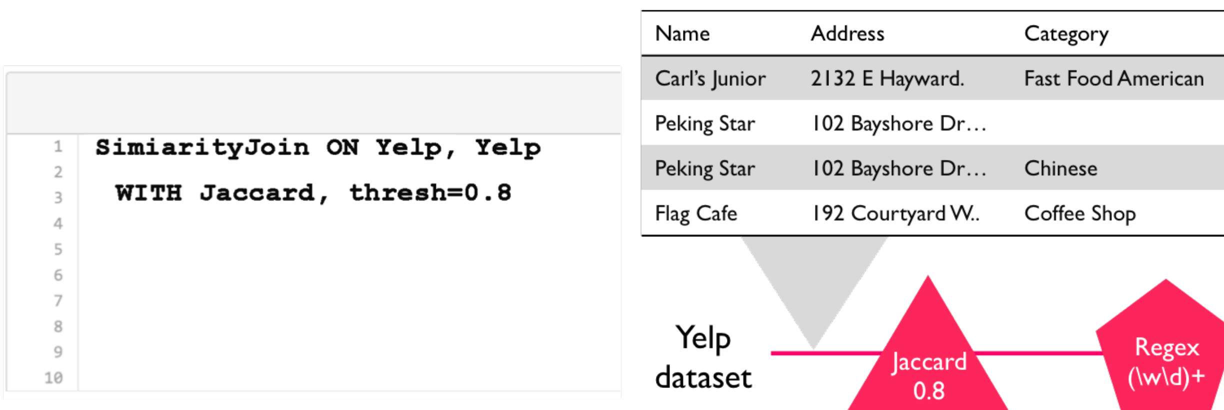


Figure 3.3: Wisteria's plan-level view.

low cost, low latency, and low accuracy. The `with` clause of the data cleaning language allows users to explicitly specify physical operators, and Wisteria's recommendation engine suggests pipeline modifications to navigate the tradeoff space.

3.3 Planning Interface

In order to enable data analysts to easily specify plans and iterate on them, Wisteria provides a planning UI. The interface is made up of two views: the plan-level view and the operator-level view.

3.3.1 Plan-Level View

The plan-level view, pictured in Figure 3.3, allows users to specify and modify the end-to-end plan. The dashboard contains both a visual plan interface and a text box to specify data cleaning operations. Using the DSL, operations are specified at a logical and physical level simultaneously. Using the visual interface, new logical operators are dragged onto the visual plan and are then instantiated as logical operators using the operator-level view described in Section 3.3.2. When the user is satisfied, she can run the plan and see the results at any stage of the plan by clicking on the relevant line segment of the visual plan.

3.3.2 Operator-Level View

To create, view, and change individual operators in the plan, Wisteria provides an operator-level view, pictured in Figure 3.4. This view lists the parameters of an operator. When a user selects a specific parameter, the system automatically displays recommendations for modifying the parameter if any are available. For example, in Figure 3.4, the system recommends switching the similarity metric used by the Similarity Join operator from Jaccard similarity to edit distance, which might be more appropriate if the join attribute does not

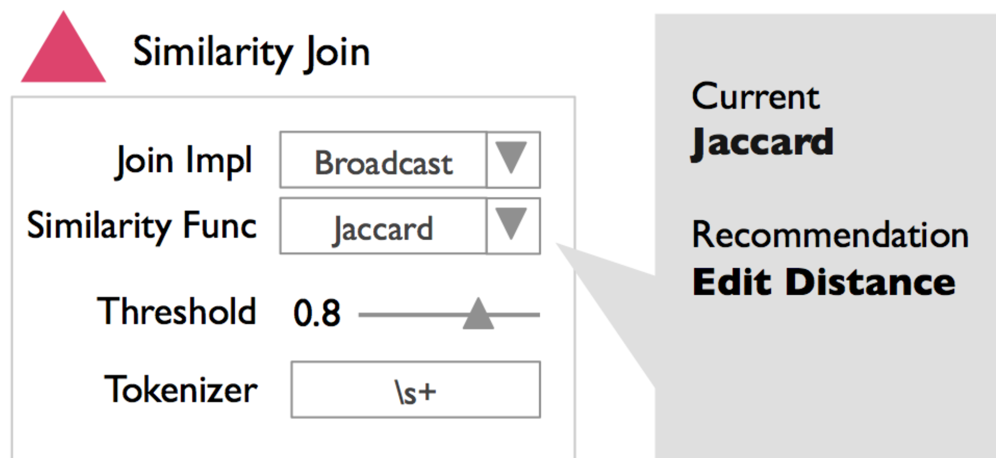


Figure 3.4: Wisteria's operator-level view.

have many tokens. Alternatively, users can manually adjust parameter values. When the operator-level view is closed, Wisteria automatically re-runs the plan if changes have been made.

3.4 Supporting Interactive Plan Generation

In order to support cleaning plan specification and evolution, Wisteria leverages three techniques: sampling, recommendation, and crowdsourcing.

3.4.1 Sampling

Prior work has explored the problem of estimating aggregate query results over dirty data [143, 87]. The SampleClean system [143] showed that aggregate queries can often be answered with very high accuracy (i.e 99%) with only a small fraction of clean data, so that it is only necessary to clean just enough for the application's data quality requirements. In Wisteria, sampling is implemented as a logical operator that can be used for quickly prototyping and optimizing workflows on samples of data and then transferring these optimizations to full datasets. I find that many important features of iterative data cleaning workflows can be posed as aggregate queries on samples with confidence intervals. To determine whether a data cleaning operation has a significant effect, a query can be used to test the effects of this operation on a sample. For example, a cleaning plan that deduplicates restaurant categories could count the number of Chinese restaurants in a sample to test if different deduplication algorithms significantly affect the count. Sampling and result estimation are salient features of Wisteria that allow for parameter tuning and recommendations for changing a workflow without evaluating all possible workflows on the full data. Next, I discuss how to efficiently generate these recommendations.

3.4.2 Recommendation

Wisteria contains a basic recommendation engine that recommends changes to a data cleaning plan based on user feedback. A user can specify a set of ground truth tuples, and our system will optimize over data cleaning plans that best reproduce the ground truth. There are three types of recommendations: (1) parameter change, (2) operator replacement, and (3) operator addition. To realize and execute the recommendations, the system uses caching and lineage to efficiently re-evaluate a workflow.

Parameter Change. Many of the physical operators in Wisteria have tunable parameters, whose values are often very dataset-specific, and the user feedback provides a way to evaluate the quality of the initial parameter choice. For example, Similarity Joins have a similarity threshold and a similarity function. Increasing this threshold reduces the selectivity of the join, and Wisteria needs to choose a threshold that maximizes accuracy. This problem can be solved by a minimum-cost spanning tree over a similarity graph (edges represent non-zero similarity) over tuples.

Operator Replacement. Wisteria recommends changes to physical operators when the user indicates that they are not satisfied with the output. For example, user feedback (e.g., indication of tuples that were not cleaned correctly) can be viewed as crowd labels and used to estimate the quality of the current operator. If the quality is low, Wisteria recommends replacing a physical operator with an active learning variant that leverages the crowd to achieve high accuracy. Additionally, Wisteria can try different variants of automated operators to test how accurate they are with respect to the user feedback.

Operator Addition. There are also cases where the system may want to add another physical operator, while still preserving the logical input-output behavior of the workflow. It is common in extraction tasks to have most tuples accurately extracted with an automated extractor but only a small subset requiring additional inspection. For these cases, Wisteria can add a crowd-based Filter operator to separate these examples for additional cleaning.

Cost Estimates. Of course, changing a plan when using crowdsourcing may significantly change its cost. For every recommendation, Wisteria estimates the number of additional tuples processed by the crowd operators and provides the user with an estimated cost. Based on a user-specified cost per task, the system can estimate the number of tasks needed to clean the dataset.

Caching allows for result re-use if a downstream operator is modified or added. If the system has sufficient memory, then it can naïvely cache all intermediate results. Otherwise, the key challenge is to select a subset of results to cache. Wisteria chooses which results to cache by integrating the caching framework with its recommendation engine. When the system makes a recommendation for a change, it caches the results of the preceding operator in order to improve the performance of evaluating potential recommendations and switching to the new plan.

Lineage allows the system to understand how results change if upstream operators are

modified. For example, decreasing a similarity join threshold increases the number of output pairs without affecting existing output pairs. The key property here is monotonicity, and some types of monotone Filter and SimilarityJoin operators are data cleaning analogs for a Select-Join relational algebra. Wisteria therefore models upstream hot-swapping as an incremental view maintenance problem and updates the final result based on the insertion or deletion of tuples earlier in the plan.

3.4.3 Crowdsourcing

As described in Chapter 2, working with crowds is inherently challenging. Unlike when using automated operators, the accuracy and speed of processing each tuple might vary widely with the crowd worker assigned to it. Completion time of an operator depends on the response times of individual workers, and on real-world crowdsourcing platforms, the distribution of response latencies is highly skewed. In contrast, in order to provide high-quality recommendations to users (and in order for plans to evaluate efficiently), Wisteria requires crowd-based operators to provide consistent output at reliable rates. Otherwise, cost and time estimates will be inaccurate, and Wisteria will make poor recommendations.

Wisteria addresses this problem by maintaining a pool of high-speed, high-quality crowd workers and developing task routing strategies that can avoid assigning tasks to slow workers and leverage redundancy to significantly reduce the time that is required to clean data with the crowd. Additionally, active learning techniques reduce the number of tuples that require crowd work to clean the data. The **Crowd Manager** component of Wisteria's architecture has evolved into the AMPCrowd system [9], which is described in detail in Chapter 4.

3.5 Conclusion

Dirty data is a fundamental obstacle to modern data analytics applications, and cleaning it requires people to play a key role. In this chapter, I introduced Wisteria, a system that supports the iterative development of data cleaning workflows. Wisteria allows the user to construct, adapt, and optimize cleaning plans with automated parameter recommendations, separating logical data cleaning operators from their physical implementations (e.g., rules, learning, or crowdsourced). A version of this system containing the core mechanisms for specifying cleaning plans, the operator API, and implementations of several physical automated and crowdsourced operators has been made open source [124].

A key insight from this work is that users cannot easily iterate on cleaning plans unless they are able to rapidly observe and evaluate the quality of their proposed modifications. To provide this functionality, the system must repeatedly evaluate slightly different plans on carefully chosen samples of data, and since many data cleaning operations use the crowd, this relies on having highly available workers processing data at interactive latencies. In the next chapter, I describe the design of a system that focuses on precisely the goal of getting reliable low-latency responses from crowd workers who may not all be sufficiently fast.

Chapter 4

CLAMShell: Speeding up Crowds for Low-Latency Data Labeling

The Wisteria system described in Chapter 3 must repeatedly process data with the crowd at interactive latencies in order to allow the user to rapidly iterate on data cleaning plans. At first glance, this seems like a nearly impossible requirement. How can human workers process data fast enough so that the system is not constantly waiting for crowd tasks to complete, especially as the size of the dataset grows? And even if crowd tasks can be accelerated to near-machine timescales, how can the system guarantee that this will occur consistently?

These questions are fundamental to the design of performant crowd systems, and in this chapter I demonstrate that it is in fact possible to answer them. As I alluded to in Section 2.4.4, much of the crowd performance bottleneck arises not because humans are unable to perform work at the necessary speeds, but because existing systems are not designed with crowd performance in mind. The gap between optimal human performance (for example, humans can complete individual image labeling or entity resolution tasks in under a second) and average-case observed behavior (tasks can take minutes to hours to complete!) provides an opportunity for optimization. CLAMShell, the system I describe in this chapter, takes advantage of that opportunity to accelerate crowd-powered data labeling by an order of magnitude and reduce its variance by two orders of magnitude.

4.1 Introduction

Though crowd-based data labeling systems seek to reduce cost and speed while maximizing quality, most research has focused only on the trade-off between quality and cost, routinely reporting task latencies on the order of minutes to hours to complete an average task [18, 84, 52]. For systems that iterate between human input and automated processing, these latencies are clearly unacceptable, as users cannot interpret the data until it has been processed.

In this chapter, I explicitly tackle the trade-off between cost and latency for crowd-sourced labeling tasks. Though there are a few existing works that explicitly aim at tackling

latency, they are either tailored to specific tasks [99, 101, 137], targeted towards a single source of latency such as recruitment time [16, 20, 55], or focused on machine learning techniques (e.g., active learning) that ignore the practicalities of live crowdsourcing and may be counterproductive in terms of wall clock latency [104].

In addition, predictability of overall task latency is an important consideration that has not been carefully studied. Depending on numerous external factors, the quantity, quality, and speed of available workers on crowd platforms such as Amazon's Mechanical Turk (MTurk) [105] can fluctuate wildly [74, 73] and result in individual task latencies from seconds to even days. I argue that the variance of task latency must be within single-digit seconds before it can be usefully embedded in interactive user-facing applications such as Data Wrangler [82].

In this chapter, I introduce CLAMShell[1], a system that speeds up crowds in order to achieve consistent, low-latency data labeling. Rather than focus on a single algorithm or step in the data labeling lifecycle, the goal is to develop a collection of pragmatic techniques to clamp down on latency and variance during all stages of labelling.

To this end, I systematically address each of the dominant sources of latency in crowd systems introduced in Section 2.4.4 (per-task latency, batch-wise latency, and end-to-end overall latency) with three novel techniques: *Straggler mitigation* uses redundant labelers to mitigate 'straggler tasks' at the end of batches, decreasing the variance of batch labeling time from minutes to fractions of seconds. *Pool maintenance* uses threshold-based eviction techniques to maintain a pool of fast, high-quality workers and decrease the average time to label each task. *Hybrid learning* combines active and passive learning to exploit crowd pool parallelism when there are more workers available than the active learning batch size, and dynamically favors passive learning on datasets where active learning performs poorly. I then evaluate CLAMShell on real MTurk workers, demonstrating up to 8$\times$ speedups in label acquisition time and over 2 orders of magnitude reduction in variance compared to typical non-optimized deployments. A key benefit of this work is that all of these optimizations are compatible with standard quality control algorithms such as redundancy-based voting schemes and worker quality estimation algorithms.

An open source implementation of CLAMShell, AMPCrowd [9], augments the techniques described in this chapter with an extensible declarative API, support for additional crowd platforms, and automated quality control. It is being used in industry and to support other crowdsourcing research projects [146].

4.2 Tackling Crowd Latency

Section 2.4.4 categorized the primary sources of crowd system latency into task latency, batch latency, and full-run latency. Table 4.1 summarizes the sources of latency described in that section, and notes with an asterisk sources that have been addressed in the literature.

[1] Crowd LAtency Mitigation Shell.

From the table, it is clear that there is ample opportunity to improve the state of crowd-sourced latency. This section reviews existing approaches to speeding up crowd systems, then describes how CLAMShell comprehensively covers all of the sources of latency.

Task Latency	**Batch Latency**	**Full-Run Latency**
Recruitment*	Stragglers	Decision Time
Qual & Training	Mean pool latency	Task Count*
Work*	Pool variance	Batch Size
		Pool Size

Table 4.1: Classification of sources of latency in data labeling.

4.2.1 Existing Literature

The first work in low-latency crowdsourcing addressed recruitment time, a dominant source of task latency. Bigham et al. [20] frequently repost tasks (among other techniques) to improve the chances of workers accepting their tasks. However, if widely adopted, such techniques would likely exacerbate recruitment time. Gao et al. [55] algorithmically increase prices over time to encourage workers to accept tasks by a deadline. Bernstein et al. [16, 17] proposed the *retainer model*, which pre-recruits a pool of crowd workers (a retainer pool) and pays them to wait until tasks arrive. In settings where tasks are streaming or come in batches, this model can effectively eliminate recruitment time at a small cost. In our work, we build on top of the retainer model.

Work time has been reduced by re-designing task interfaces [99]. For example, Marcus et al. [101] study join interfaces for images, and design interface batching techniques that let workers complete up to 9 pair-wise comparisons in the same time as a single pair-wise comparison task. However, these approaches are task specific, so CLAMShell views them as complementary to its general task optimization framework and does not explicitly address them.

Finally, algorithmic analysis and machine learning have been used to reduce task count. The former focus on efficient algorithms for specific operations (e.g., entity resolution [142], counting [99], or information retrieval [41, 111]). These focus on full-run latency, and could leverage CLAMShell's per-task, per-batch, and machine learning techniques.

The latter fits models to data from already completed tasks, then uses those models to predict remaining task outputs once the prediction quality exceeds a user-defined threshold. In this setting, active learning [36] is a commonly used method [104, 58]. Given unlabeled data, active learning iteratively uses a point selection algorithm to pick a small set of informative points to acquire labels for, and incorporates the new labels into its model. The algorithm continues until the model accuracy (evaluated via cross-validation, e.g.) converges. Active learning is indispensable when there are more items than can be practically labeled, and can be used in conjunction with algorithmic approaches that rely on the labels [41, 142].

Despite reducing the task count, active learning may counter-intuitively *increase* the overall latency by constraining the parallelism due to its batch size limitations. Its convergence properties have only been proved when the batch size is 1. and larger batch sizes (e.g., 10) have only been tested empirically. When the number of workers significantly exceeds the batch size, active learning can be much slower than labeling as many random tasks in parallel as possible and using a passive learner.

4.2.2 Towards a Comprehensive Solution

Improving the performance of data labeling requires a trade-off between cost, accuracy and latency: a user wants to label a set of items accurately at high speed and low cost. This work focuses especially on reducing latency by sacrificing cost. To this end, CLAMShell systematically tackles the primary sources of latency (Table 4.1) in a general purpose labeling system:

1. Task Latency: CLAMShell addresses task latency by adopting retainer pools to reduce recruitment costs. CLAMShell automatically maintains the pool size at p as workers abandon the pool, and provides guidance about how the cost and latency will be affected by changing p. In addition, CLAMShell trains and verifies worker qualifications as part of recruitment, ensuring that every worker in the pool is immediately available to provide useful work when new tasks arrive.

2. Batch Latency: *Straggler mitigation* uses worker redundancy on slow tasks to compensate for long-tail latencies. *Pool maintenance* selectively replaces pool workers to progressively shift and tighten the latency distribution towards faster responses. Together, they eliminate straggler effects, reduce mean pool latencies over time, and significantly reduce batch variance.

3. Full-Run Latency: CLAMShell uses a hybrid strategy that allocates subsets of the worker pool to active and passive learning. In addition, CLAMShell pipelines the expensive model retraining and uncertainty sampling steps with crowd labeling to eliminate decision latency at the cost of slightly stale model results.

Note that CLAMShell does not explicitly address *work time* or *pool size*: work time is often specific to the task interface, which we view as an orthogonal interface optimization problem, and pool size is typically set by operational constraints. Chapter 5 considers the setting where pool sizes can be dynamically adjusted to improve performance. Instead, this chapter focuses on the other sources of latency listed in Table 4.1.

4.3 The CLAMShell System

In this section, I present an overview of CLAMShell, a system for fast label acquisition.

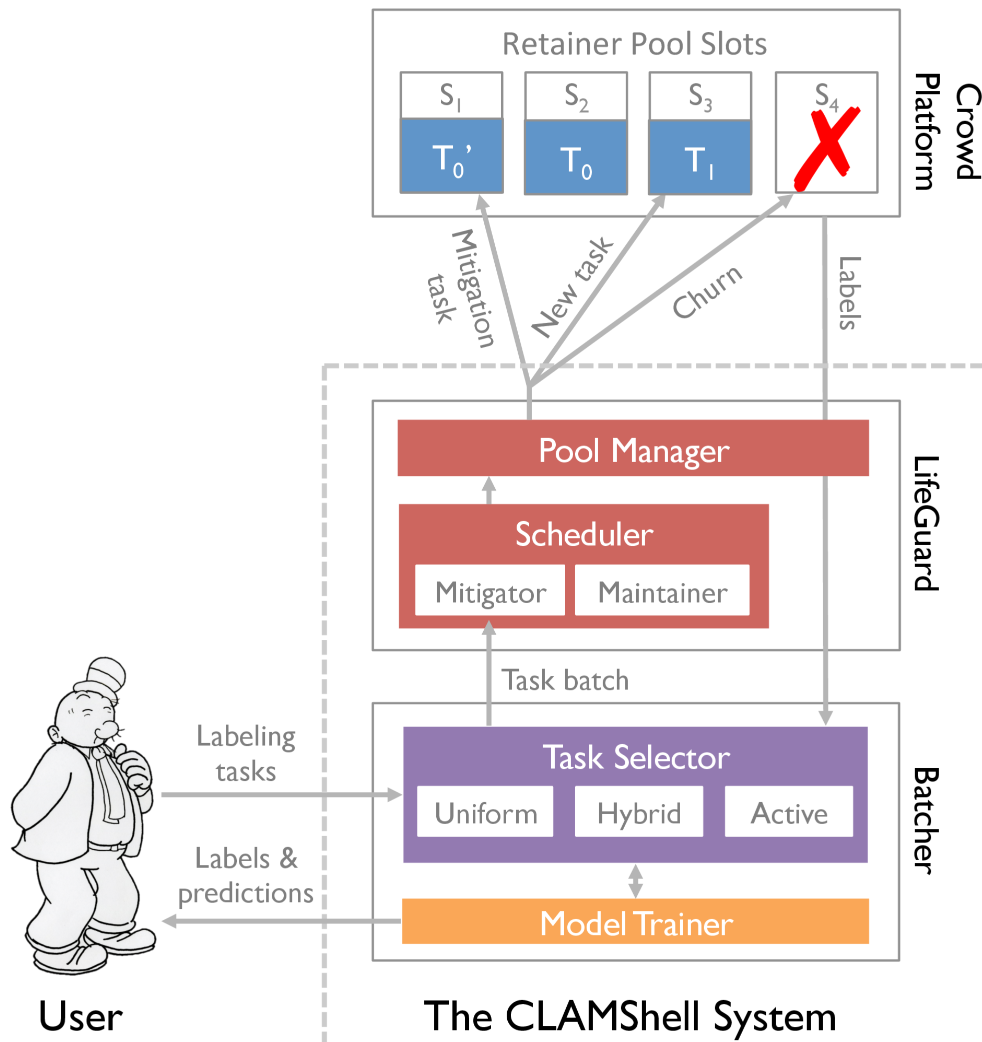


Figure 4.1: CLAMShell architecture diagram.

The CLAMShell architecture is illustrated in Figure 4.1. The user submits a set or stream of labeling tasks to the *Batcher* and uses the *Task selector* (Section 4.5.1) to pick B incomplete tasks to process in the current iteration. The tasks are selected via uncertainty sampling using the most recently trained model to pick tasks that benefit active learning, and random sampling to pick tasks for passive learning. The resulting batch is sent to *LifeGuard*, which schedules tasks within the batch to be sent to the *Crowd Platform*. This level of indirection is necessary when the batch size exceeds the size of the retainer pool, and so the *Mitigator* can control redundancy when there are slow tasks.

The *Crowd Platform* holds a set of slots ($S_1 \ldots S_4$) in the current retainer pool. Each slot corresponds to a persistent *retainer task* that a crowd worker has accepted, and may be empty (e.g., S_4) or contain a task (e.g., T_0). The *Scheduler* immediately sends new

tasks to available slots (e.g., S_3). If all tasks have been sent, then the *Mitigator* sends duplicate (mitigation) tasks for slow, incomplete tasks (e.g., S_1). If a slot is consistently performing slowly, the *Maintainer* may recruit and train a worker for a replacement slot in the background, then evict the slow slot (✗ in S_4).

Completed labels are sent directly to the *Batcher*, which retrains the machine learning model. The *Task Selector* uses different sampling algorithms such as uniform sampling, active learning-based uncertainty sampling, or our hybrid sampler, to pick the next batch of tasks. At all times, the user can access the completed labels and query the currently trained model for new predictions. The following example illustrates the use of CLAMShell in practice.

Example 4.1 *Consider a news outlet that is covering a live political debate, and wants to monitor and visualize the public's reaction to candidates' comments on hot-button issues by analyzing the sentiment of related tweets. Because automated sentiment analysis techniques on tweets are often inadequate [2], the company asks a crowd to label tweets as "positive", "negative", or "neutral". If the system suffered from high crowd latency, the sentiment visualization would be unable to keep up with the changes in public opinion as the debate proceeded, rendering the tool unhelpful.*

CLAMShell can be used to address this issue with both per-batch and full-run optimizations. The per-batch optimizations, including straggler mitigation and pool maintenance techniques, are designed to reduce the time that is required to label a batch of tasks using crowds, e.g., asking for crowd labels for a batch of ten tweets. Once the company has enough labeled data, they hope to switch to an automated process in the long term. The full-run optimizations, including hybrid learning, are designed to reduce the number of iterations that a learning model needs to converge, i.e., the total number of batches that crowds are asked to label.

4.4 Per-batch Latency Optimization

Per-batch optimizations aim to reduce the latency for a single batch of labeling tasks — the *Batcher* sends a batch of tasks to a pool of workers, and waits for the batch of work to complete. In this model, the dominant costs are due to the variability of worker latencies within the pool, as well as the variability within the tasks that a single worker performs.

For example, Figure 4.2 shows the CDFs of per-worker mean and standard deviation latencies from a real crowd deployment that ran $\sim$60,000 tasks to label medical publication abstracts. The figure reveals that average worker speeds are spread out from tens of seconds to hours. In addition, even workers who are very fast on average ($\sim$1 minute) can take as long as an hour or more to complete some tasks. This variation is bad for per-batch latency because the batch must block until all of its tasks are complete.

So in order to reduce per-batch latency, a system must reduce both the mean of the latency distribution (workers who are slow on average) and its variance (workers who are

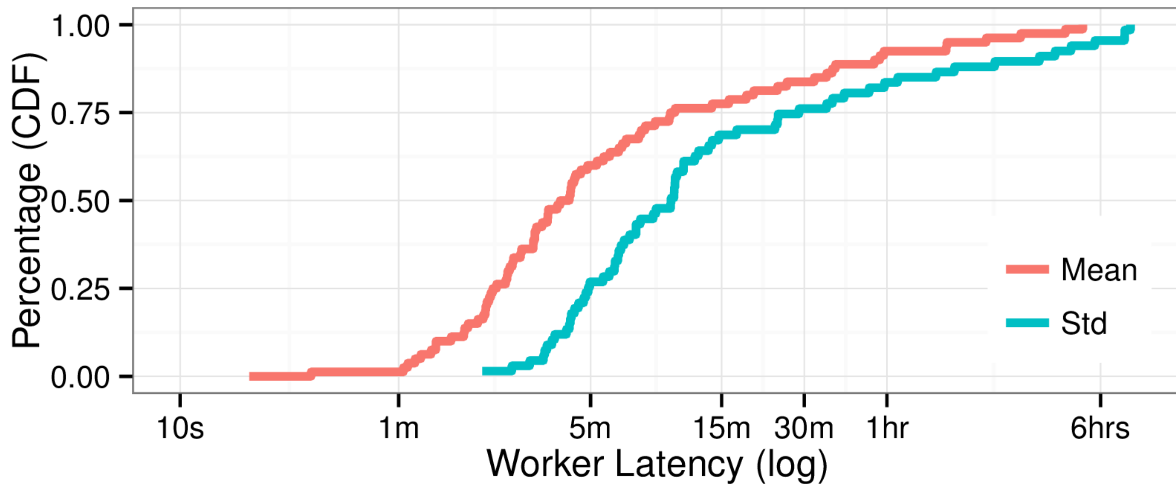


Figure 4.2: Distribution of worker latencies.

inconsistent). This section describes the mechanisms for each of these approaches respectively, along with mathematical models and simulation results. In Section 4.6, I evaluate these strategies on a live deployment on MTurk.

Throughout the following sections, references are made to experiments run in simulation. The simulated experiments are run on a python simulator that models a retainer-pool crowd data labeler and implements uncertainty sampling on top of scikit-learn's model training [113]. To simulate crowd workers, I use traces from the medical deployment described in Section 2.4.4. From each trace, the simulator measures each worker's mean labeling latency μ_i, variance in labeling latency σ_i^2, and mean accuracy λ_i. It then simulates a worker's latency on an assigned labeling task by drawing a sample i.i.d from $\mathcal{N}(\mu_i, \sigma_i^2)$, and generates the label itself by returning the correct label with probability λ_i and the incorrect label with probability $1 - \lambda_i$. Using these worker pools, the simulator can model recruitment (adding random workers to the pool), pool maintenance (releasing workers with high observed μ_i from the pool), straggler mitigation (assigning multiple simulated workers to the same task and returning the minimum of the sampled latencies), and active learning (using simulated workers to label batches of points and measuring the latency of the whole batch).

4.4.1 Straggler Mitigation: Reducing Variance

In cluster computing frameworks such as Hadoop [68] or Spark [147] where the presence of straggler tasks in a stage (e.g., reduce stage of MapReduce [43]) can delay downstream computation, replicating the slow tasks [10, 43, 11, 148] via speculative execution or task cloning [10] is an effective counter-measure.

We take a similar replication-based approach to human stragglers in our crowd pool. We

call a worker *active* if she is currently working on a task, and *available* otherwise. Similarly, a task is either *active*, *complete*, or *unassigned*. By default, CLAMShell routes only unassigned tasks to available workers until all tasks are complete. Once all tasks are active or complete, available workers must wait until the next batch to receive a task. With straggler mitigation, in contrast, such workers are immediately assigned active tasks, creating duplicate assignments of those tasks. CLAMShell returns the first completed assignment of a task to the user and immediately reassigns all other workers still working on that task to a new unassigned or active task (though it pays them for their partial work on the old task regardless). The effect of straggler mitigation is that when an inconsistent worker takes a long time to complete a task, the system hides that latency by sending the task to other, faster workers. As a result, the fastest workers complete the majority of the tasks and earn money commensurate with their speed. For example, in the medical publication abstract labeling deployment described above, the fastest worker ($\mu = 28.5$ seconds) could complete, on average, 8× as many tasks as the median worker ($\mu = 4$ minutes).

Simulation. A natural question arises when performing straggler mitigation: which task should be assigned to an available worker? In simulation, I tested several straggler routing algorithms, including routing to the longest-running active task, to a random task, to the task with fewest active workers, or to the task known by an oracle to complete the slowest. Surprisingly, the selection algorithm didn't affect end-to-end latency, and random performed as fast as the oracle solution because the fast workers complete almost all of the tasks in the batch.

A second question is: at what batch sizes is straggler mitigation effective? This can be studied in simulation by varying the pool size to batch size ratio $R = \frac{N_{pool}}{N_{batch}}$ using the random selection algorithm and different pool sizes. The benefit of straggler mitigation comes from its ability to remove the overhead of slow workers at the end of a batch of tasks. When R is higher, each batch gains the full benefit of straggler mitigation and completes at the speed of the fastest workers, however the number of tasks completed in each batch is lower. Conversely, with a small ratio, workers spend most of their time working on unassigned tasks, and the impact of straggler mitigation is lessened.

Impact on Crowdsourcing Systems. Straggler mitigation is a general technique that does not affect the programming interface of the system it is applied to. It can therefore be used easily in conjunction with any existing crowdsourcing system that processes batches of microtasks. One important benefit of hiding the variance in worker latencies is that task completion times become much more predictable. This characteristic is vital to the development of declarative crowd systems such as crowdsourced query processors, because optimizers need to be able to accurately estimate the cost of executing a declarative crowd workflow.

4.4.2 Pool Maintenance: Better Mean Latency

Straggler mitigation reduces the variance of task latencies, but if many workers in the labeling pool are slow on average, variance reduction will be ineffective at reducing per-batch latency. To improve the average speed of the pool over time, CLAMShell uses *pool maintenance*, a technique that continuously replaces slow workers in order to converge to a pool of mostly fast workers. Because a fast pool will label each task more quickly, pool maintenance reduces per-batch labeling latency over time.

The pool maintenance algorithm takes as input a latency threshold PM_ℓ, and continuously releases workers slower than the threshold asynchronously as labeling proceeds. To do so, it computes an empirical latency for each pool worker based on the worker's completed tasks and flags the worker as a candidate for removal if his latency is significantly above PM_ℓ (determined using a one-sided significance test).

Instead of removing a slower worker before recruiting a replacement, CLAMShell continuously recruits and trains workers in the background in order to maintain a reserve of new workers. Although this might seem costly, pipelining recruitment means that pool maintenance can proceed without blocking on worker recruitment, and empirically, the latency savings of pool maintenance translate to cost savings that overwhelm the cost of background recruitment (Section 4.6.2). The removed worker is paid for their active job (if any), and informed that there are no more tasks available for the experimental run. They are not blacklisted, so that future experiments are not biased.

Pool speed convergence. The following analysis derives the mean latency to which a maintained pool will eventually converge. Assume a population of workers with mean latencies μ_i distributed according to some global distribution $\mathcal{W}$, and sample an initial pool $\mathcal{P}_0 \subset \mathcal{W}$ uniformly at random from $\mathcal{W}$. Let PM_ℓ be a latency threshold splitting the distribution $\mathcal{W}$ into two parts, with probability densities q and $1-q$ above and below PM_ℓ respectively. Further, let μ_f be the mean latency among fast workers having $\mu_i < PM_\ell$, and let μ_s be the mean latency among slow workers having $\mu_i > PM_\ell$.

Then the initial pool has a mean latency $\mathbb{E}_{\mathcal{P}_0}[\mu_i] = (1-q)\mu_f + q\mu_s$. If at each maintenance step, all slow workers having $\mu_i > PM_\ell$ are removed and replaced with workers drawn randomly from $\mathcal{W}$, and letting $\mathcal{P}_i$ be the pool after i steps, it can be seen that $\mathcal{P}_1$ has mean latency $\mathbb{E}_{\mathcal{P}_1}[\mu_i] = (1-q)\mu_f + (q(1-q)\mu_f + q^2\mu_s)$, and in general $\mathcal{P}_n$ has mean latency:

$$\begin{aligned}\mathbb{E}_{\mathcal{P}_n}[\mu_i] &= (\sum_{i=0}^{n} q^i)(1-q)\mu_f + q^{n+1}\mu_s \\ &= (1-q^{n+1})\mu_f + q^{n+1}\mu_s.\end{aligned}$$

An important observation is that $\lim_{n\to\infty} \mathbb{E}_{\mathcal{P}_n}[\mu_i] = \mu_f$, that is, the pool converges to the mean latency of all workers below PM_ℓ. This implies that it is desirable to set PM_ℓ as low as possible: in practice, setting the threshold too low leads to thrashing, as I show in section 4.6.2.

Simulation. I simulated how pool maintenance affects batch latency with respect to the task to pool size ratio R using a latency threshold PM_ℓ of one standard deviation below the mean. After each batch, all workers slower than PM_ℓ are replaced with new samples from the worker distribution. With pool maintenance, the batch latency falls quickly, nearly halving in just 15 to 20 batches. When there are many more tasks than pool workers, the effect becomes less pronounced, because there are enough tasks that slow workers who only complete a small fraction of tasks do not impact the per-batch latency.

To better understand how the distribution of mean worker latency is changing over time, I measure the mean pool latency (MPL) of the worker pool over time with and without maintenance, and compare the MPL to the mathematical model's predictions. With maintenance, the pool's MPL converges quickly to the model's predicted asymptote, following the model closely across pool-size to task ratios R.

Latency Threshold. The pool maintenance latency threshold determines which workers are slow and should be removed from the pool. To pick a good threshold, the system can observe the empirical distribution of the mean latencies of all workers ever seen, and estimate the threshold as k standard deviations below that mean. The goal is to find a threshold low enough to decrease average pool latency by releasing slow workers, but high enough to avoid discarding the fastest workers from the pool. Section 4.6.2 contains experiments showing that varying the threshold and has significant impact on the benefits of pool maintenance.

Extensions. As described, pool maintenance is focused only on reducing the mean latency of the pool. However, it can be easily extended to optimize for other criteria by choosing an objective function other than worker speed. For example, using a quality metric (e.g., inter-worker agreement [23]) would converge to a high-quality worker pool, using a weighted average of quality and latency could trade off the quality and speed of the pool, and other metrics could optimize for other desired worker properties such as worker variance. Ramesh et al. [118] take a similar approach to identifying high-quality workers, though they use an oracle for accuracy and evaluate their technique only in simulation.

4.4.3 Combining Per-Batch Techniques

Interference Effects. Both straggler mitigation and pool maintenance deal with tail latencies — maintenance detects and removes workers whose average speeds are outliers, and straggler mitigation hides individual workers' outlier tasks. Unintuitively, in initial live experiments, I found that naively combining the two techniques together resulted in zero or even negative gains as compared to straggler mitigation alone. For example, the average number of workers replaced in each batch was reduced from ~ 30 to less than 5 despite similar worker distributions.

The reason is that straggler mitigation terminates slower tasks, artificially skewing every worker's completion times towards the latency of the fastest workers. This makes directly

measuring true worker latency infeasible. In response, I developed a simple model called `TermEst` to estimate the average latencies of terminated tasks based on the number of times a worker's task is terminated.

Assume the worker pool is represented by two worker types — slow workers w_s and fast workers w_f that take latencies of $l_{s,j}$ and $l_{f,j}$ to complete task t_j respectively. The goal is to estimate the latency of w_s' terminated tasks. I adopt this model again in Chapter 6 to simplify the analysis of adaptive algorithms for finding fast workers. Let w_s start N tasks $T = \{t_1, \ldots, t_N\}$, where $T_t \subseteq T$ are terminated, and $T_c = T - T_t$ are completed. Let $l_{k,T} = \frac{1}{N}\sum_{t_i \in T} l_{k,i}$ be the average latency for worker w_k to complete a random task in T, and let l_k be w_k's true mean latency. Since w_f can start working on t_j at any time after w_s with uniform probability, the probability that w_f starts early enough to cause w_s to terminate is $\frac{l_{s,j} - l_{f,j}}{l_{s,j}}$. Thus, w_s is expected to be terminated N_t times over N tasks:

$$N_t = \sum_{t_i \in T_t} \frac{l_{s,i} - l_{f,i}}{l_{s,i}} \approx \frac{l_{s,T_t} - l_f}{l_{s,T_t}} \times N.$$

Rearranging the terms and using $N_c = N - N_t$, l_{s,T_t} can be estimated as:

$$l_{s,T_t} = \frac{l_f \times N}{N_c}.$$

In order to compensate for the lack of latency evidence when N is small and avoid divide-by-zero errors when all of a worker's tasks are terminated ($N = N_t$), I add a smoothing term α to N. In practice, l_f is estimated as the empirical mean of the workers that caused any of w_s' past jobs to terminate:

$$l_{s,T_t} = \frac{l_f(N + \alpha)}{N_c + \alpha}.$$

Finally, the overall latency of w_s can be estimated by taking the the weighted average of l_{s,T_t} and the empirical mean latency of the tasks w_s is able to complete, l_{s,T_c}:

$$l_s = \frac{N_t}{N} \times l_{s,T_t} + \frac{N_c}{N} \times l_{s,T_c}.$$

Note that this formulation is equivalent to modifying the latency threshold on a *per worker* basis. Thus, while changing the global latency threshold is important for setting a worker replacement rate, this adjustment replaces workers who are frequently terminated.

Working with Quality Control. One potential drawback of this approach is that by selecting workers based only on speed, these techniques might reward spammers or other fast but inaccurate workers. Luckily, empirical data from the experiments in this chapter show that faster workers are no more likely to be inaccurate than slow workers, but it is worth mentioning that traditional quality control techniques are entirely complementary to these per-batch techniques. For example, CLAMShell implements redundancy-based quality

control algorithms such as [75] or [83] that use votes from multiple workers to better estimate the true answer by simply acquiring more labels using straggler mitigation and pool maintenance. This might appear confusing, as straggler mitigation also acquires extra labels, but there is a key difference: straggler mitigation *asks* for extra labels but doesn't wait for them, whereas quality control *requires* extra labels and waits for all of them.

4.5 Full-Run Latency Optimization

In order to eliminate the need to manually label all points in a potentially large set, CLAMShell acquires labels for only as many points as needed to train a predictive model of sufficient quality, then uses that model to impute labels for all remaining points. As described in Section 4.2, there are many factors that influence the latency of the labeling process. Relying on learning greatly decreases the *task count* necessary to label the entire dataset, but has implications for the *decision latency* and *batch size* involved. In particular, CLAMShell uses uncertainty sampling to reduce the task count even further, but trades this improvement for increased decision latency (the learner must choose which points to label next) and decreased batch size (active learning is inherently iterative and cannot label as many points in parallel).

In this section, I describe how CLAMShell ameliorates the drawbacks of active learning for low-latency labeling. I introduce *hybrid learning*, a novel technique which combines active and passive learning to maximize pool parallelism and hide the inherent limits of active learning batch size. I also describe how CLAMShell leverages existing techniques to set an effective batch size for active learning and uses asynchronous model retraining to hide active learning's decision latency.

4.5.1 Hybrid learning

Active learning uses the current trained model to decide which points to label in the point selection phase, reducing the number of points needing labels in order to train a high-quality model. In practice, however, there are two major challenges to active learning at low latency. First, at each iteration, active learning has a limited batch size—setting the batch size too high can cause the model to converge even more slowly than passive learning [126]. This limits the wall-clock speed at which active learning can proceed. Second, when labeling work is challenging, it will be hard to train a good model. As a result, the current trained model may misguide the point selection phase, and active learning may perform poorly, perhaps even worse than passive learning. On the other hand, passive learning that trains a model using a randomly sampled data points can proceed as fast as the crowd can label, but it will waste human effort for easy labeling work.

To address these issues, I propose *hybrid learning* in CLAMShell, with the basic idea of preserving the best traits of both passive and active learning, allowing for fast model convergence on both easy and hard data labeling work. Hybrid learning simultaneously

acquires labels using the active selection strategy and random sampling, maximizing crowd worker parallelism and compensating for datasets where active learning alone would perform poorly. As a result, label acquisition can proceed at high speed in spite of a low active learning batch size.

Point Selection. Once a batch size has been selected for active learning (Section 4.5.2, below), hybrid learning attempts to maximize crowd worker parallelism by ensuring that each worker in the pool has at least one point to label. That is, given a batch size k and a pool size p, hybrid learning uses the active selection criterion to choose k points for labeling, then randomly selects $\max(0, p-k)$ points for passive labeling. Because CLAMShell caches all previously labeled points, if the points chosen for active or passive labeling overlap, their labels are read from the cache and additional points are selected for labeling.

Model Retraining. Once a new batch of points has been labeled, hybrid learning retrains a model on all previously observed labels. These points come from two sampling distributions: uncertain sampling (active learning) and random sampling (passive learning). Currently, CLAMShell retrains the model on the union of these points without distinguishing between them, though it does weight points based on the active-to-passive ratio (i.e., $\frac{k}{p-k}$). If users provide hints to CLAMShell about how hard their labeling work is (e.g., very difficult), CLAMShell can adjust these weights accordingly.

4.5.2 Active learning batch size

Because the speed of active learning is constrained by the size of its batches, setting a good batch size is important for fast convergence. Too small, and training will be slow because it will take a long time to label all the points. Too large, and training will be slow because each batch contains less useful points, slowing down convergence to a good model (or even converging to a bad one!). The literature provides no guidance on an appropriate batch size for batch-mode active learning, assuming that that the batch size is chosen by the user in advance. Chakraborty et. al [27] offer an active learning technique that dynamically sets the batch size, but it is not generic across learners and requires knowledge of the labeling time for each instance. I experimented extensively with the active learning batch size, and found that once batch size was within a reasonable range (10-40), there was no significant correlation between batch size and convergence rates on any single dataset, let alone across datasets.

As a result, CLAMShell relies on empirical results from the hybrid learning experiments in Section 4.6.5 to set an active learning batch size that works well with our hybrid strategy. Those experiments show that the fraction of the pool $r = \frac{k}{p}$ allocated to active learning has a significant impact on the convergence of the learner, and that $r = 0.5$ is a reasonable value for multiple datasets. In end-to-end experiments, I set $k = 0.5p$ accordingly.

4.5.3 Active learning decision latency

The time taken by the active learner to retrain a model and select a new batch of points after the previous batch has been labeled has a significant impact on full-run latency, because the labeling process blocks until the learner is ready with the next batch. To mitigate this latency (which is not an issue for passive learning), CLAMShell uses two known techniques.

First, rather than consider all unlabeled points for selection in the next batch, we consider only a uniform random sample of the points. This has been shown to have little impact on active learning convergence, and offers significant performance improvements: the point selection time is linear in the sample size, not the size of the entire unlabeled dataset, which might include millions of examples.

Second, rather than performing retraining and selection synchronously at the end of each batch, CLAMShell continually retrains models asynchronously on the latest available points. A new batch of points is selected based on each new model, so at any point in time there is an available model and an available selection of points for the next batch. When each batch of points completes the labeling process, the next batch is selected based on the most recently computed model. This trades off decision latency for staleness of points to be selected, and empirically I find that it does not significantly impact model convergence.

4.5.4 Putting it all together

CLAMShell is powered by three novel techniques: Straggler Mitigation (straggler), Retainer Pool Maintenance (pool), and Hybrid Learning (hybrid). Table 4.2 summarizes their impact on system performance across four axes: (1) Can they improve the mean latency of labeling? (2) Can they mitigate the variance of individual workers' labeling latency? (3) Do they require additional cost to use? (4) Are they general or restricted to a certain labeling setting?

CLAMShell	**Latency**		**Cost**	**General**
Technique	**Mean**	**Variance**		
straggler	Yes	Yes	Increase	Yes
pool	Yes	Yes	No Change	Yes
hybrid	Yes	No	Increase	AL

Table 4.2: CLAMShell techniques (AL: Active Learning).

4.6 Evaluation

In this section, I evaluate CLAMShell both in simulation and on live crowd workers on MTurk in order to show that it enables data labeling to proceed at interactive speeds. After evaluating each technique in isolation, I provide end-to-end experiments demonstrating the

Param	Description
PM_ℓ	Latency threshold for pool maintenance
SM	Straggler mitigation: on (SM), off ($NoSM$).
N_p	Number of workers in the retainer pool.
N_g	Task size: # records grouped a HIT.
	Small (1), Medium (5), Large (10 records)
R	Pool-batch ratio.
Alg	Learning algorithm: active (AL), passive (PL),
	hybrid learning (HL), or none (NL)

Table 4.3: Experimental Parameters

total time it takes to label unlabeled datasets. Table 4.3 summarizes important parameters varied in the experiments.

4.6.1 Experimental Setup

Live Experiments. The live experiments discussed below were run on AMPCrowd [9], a custom implementation of the retainer model for MTurk. Recruitment occurs by repeatedly re-posting recruitment tasks every 3 minutes to MTurk until the desired number of workers have joined the pool. Workers are paid $.05 / minute to wait for available work once they join a pool, and $.02 / record to perform the work once it becomes available. MTurk tasks require a minimum qualification of 85% worker approval to join a pool. Experiments in these pools were run at multiple times of day on both weekdays and weekends. In contrast with prior work, I found that results were remarkably consistent across these parameters when using our latency mitigation techniques. This may be the result of the relatively strict qualification requirement, or may reflect more systemic changes in the MTurk marketplace. Following the retainer pool model, the experiments assume recruitment time is amortized across batches and measure latency from the moment the first task is sent to the pool, rather than from the beginning of the recruitment process. Overall, I collected timing results for nearly 250,000 individual task assignments over the span of several weeks.

Datasets. The active learning tasks run in this evaluation are all classification tasks, based on publicly available datasets. The MNIST dataset [93] contains 70,000 black and white images of handwritten digits, and the multi-class classification task is to detect which digit is in each image. Raw pixel values were used as features, leading to 784 features per image, and entropy was used as the uncertainty metric when performing active learning. The CIFAR-10 dataset [88] contains 60,000 color images of various objects, and the classification task is to identify the category of the primary object in each image. In order to make the learning task simpler, the number of topic categories was limited to two: "Birds" and "Airplanes". Again, raw pixel values were used as features, generating 3072 features per image. In addition to

the real datasets, which have concrete labeling tasks that can be sent to human workers, the scikit-learn data generator, which builds classification problems following the algorithm from [62], was used to generate datasets of varying difficulty to illustrate the relationship between problem hardness and the performance of our techniques.

4.6.2 Pool Maintenance

In this section, I evaluate the effects of pool maintenance on batch time. The experiments execute 500 tasks that label MNIST digit images. I compare tasks of varying size (Small, Medium, Large) that use $N_g = 1, 5,$ or 10 MNIST images, respectively. The latency threshold is set to $PM_\ell = 8$ and $PM_\ell = \infty$ (no maintenance).

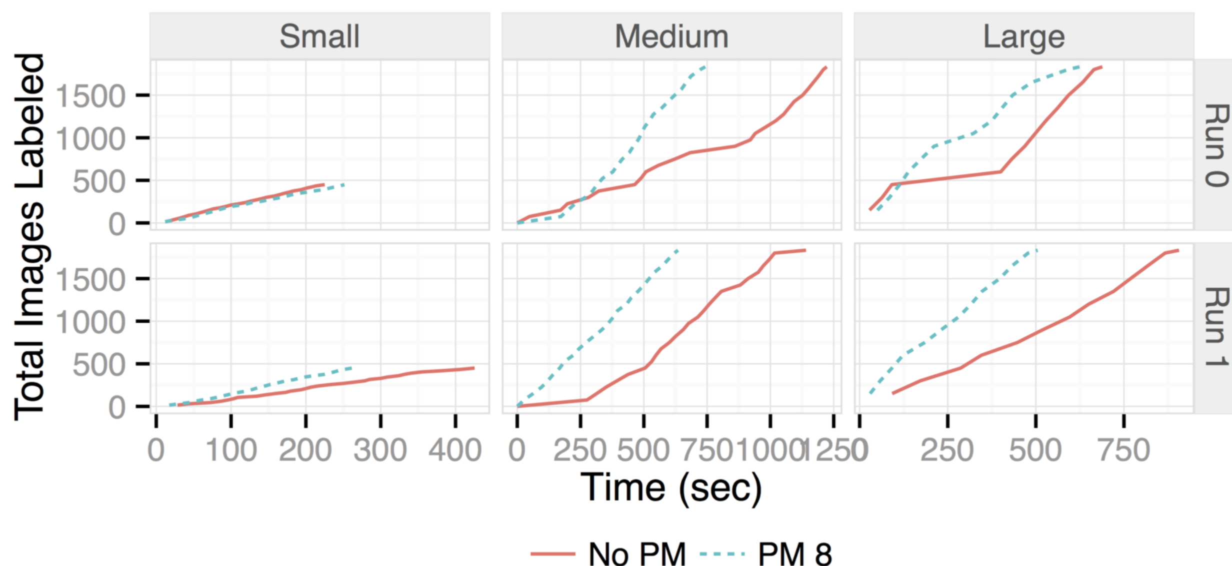


Figure 4.3: # points labeled over time in 2 typical runs.

Figure 4.3 is an overview of the total number of labeled points ($N_g \times N_{tasks}$) over time for each configuration. The slope of each curve describes the speed of task completion, where a flat curve denotes stragglers that take a very long time to complete a task. Task completion for smaller tasks is uniformly fast, so pool maintenance provides little additional benefit; however, larger tasks are affected by outliers, and maintenance's ability to cull slow workers helps reduce the presence of very long tasks.

Overall. Ultimately, pool maintenance does not improve end-to-end latency for small tasks significantly, but is able to reduce the latency for medium and large tasks by 1.3× and 1.8× on average, respectively (Figure 4.4). Interestingly, despite the added cost of recruiting workers concurrently with labeling tasks, pool maintenance is able to reduce the overall cost

Figure 4.4: Summary of end-to-end cost and latency experiments with and without pool maintenance.

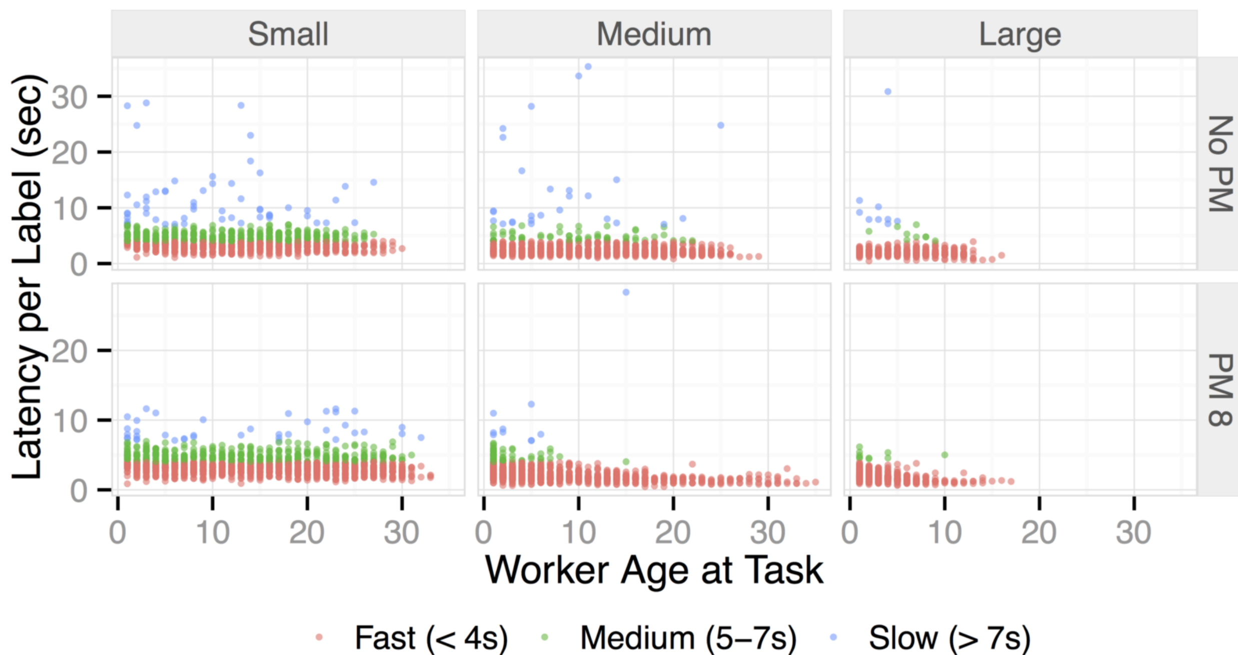


Figure 4.5: Comparison between age of the worker in the pool when starting a given task and the time to complete the task. Tasks where the latency per labeled point is greater than 8 seconds are colored in blue.

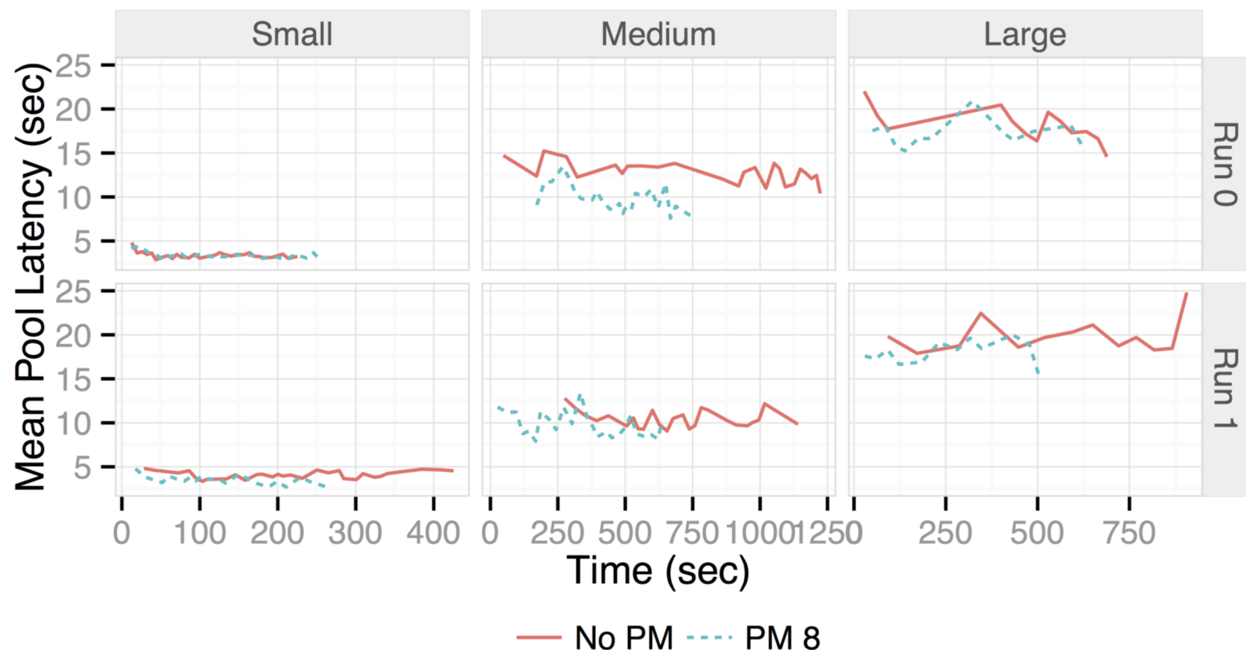


Figure 4.6: Mean pool latency over time.

of the medium and large tasks by $7-16\%$. This is due to finishing the experiment faster and saving the cost of paying workers to stay in the retainer pool. Changing the rate paid to waiting workers may increase or reduce this effect.

Latency Distribution. To better understand how pool maintenance effects the composition of the worker pool, Figure 4.5 plots task completion speeds against the age of the worker when starting a given task. A worker's age is defined with respect to task t_i as the number of tasks the user has already completed in the experimental run. The y-axis shows the latency to acquire a single label, computed as $\frac{\text{task latency}}{N_g}$; each column shows all tasks across the runs for a given task size; and the top and bottom rows are with maintenance turned on (PM_8) and off (PM_∞). In addition, the points are categorized as fast (<4 sec per label), medium ($5-7$ sec), or slow (≥ 8 sec). Although workers that are new to the worker pool naturally exhibit high task latency variability, maintenance is able to purge the slow workers over time. For every task size, the slow and even medium latency tasks are nearly all removed once workers have remained in the pool for more than 4 minutes. In contrast, the lack of pool maintenance allows slow and highly variable workers continue working on tasks, so that slow tasks are seen throughout the entire experiment.

Mean Pool Latency. Figure 4.6 provides a different view on pool maintenance's effects on the worker pool – it measures the *mean pool latency (MPL)* for each batch of tasks sent to the

pool throughout the experiment. MPL is computed as the average latency of all completed tasks in the pool. Each subplot compares the MPL with and without maintenance for a given experimental run and task size. While the average of each pair of curves is similar, pool maintenance shows significantly less variance across the batches because it effectively removes the long tail of the latency distribution. The variation in the pool maintenance curve is simply due to the variation of the newly recruited workers.

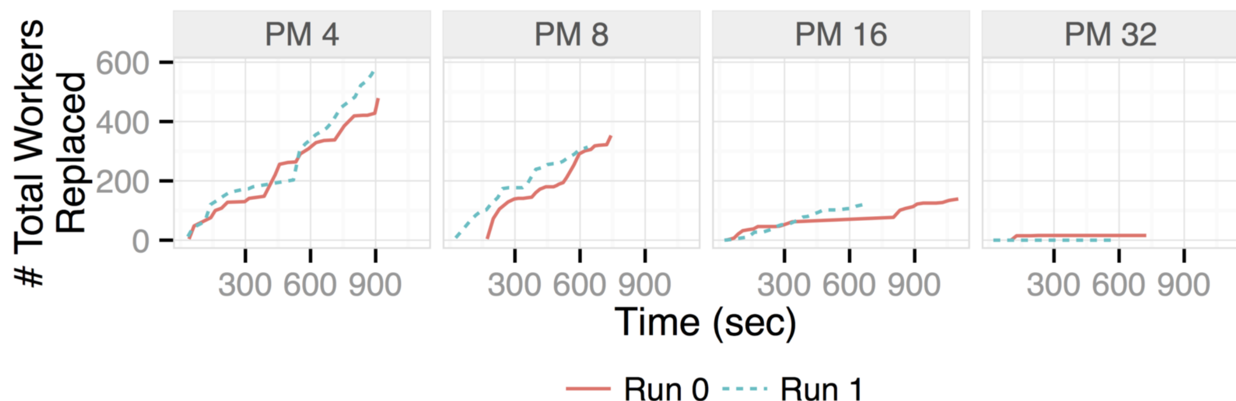


Figure 4.7: The number of workers replaced over time for varying maintenance latency thresholds.

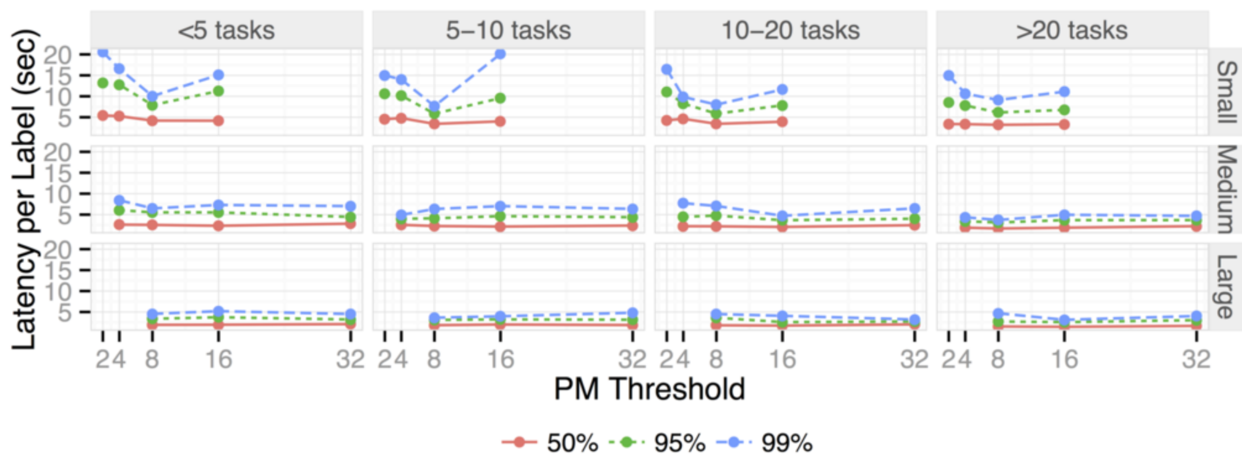


Figure 4.8: 50th, 95th, and 99th percentiles of task latency as maintenance latency threshold varies. Each facet is a different amount of time into the experiment.

Latency Threshold. The analysis of MPL shows that pool maintenance is able to remove outliers from the worker pool. However, the reduction in MPL is not as fast as predicted

by the model or simulations presented in Section 4.4.2. This is expected, as workers may not maintain consistent speed over time, and empirical estimates of worker's speed may be inaccurate. Another potential issue may be that the latency threshold is poorly tuned, so in our final experiment (Figures 4.7 and 4.8), I study whether varying the latency threshold between 2 and 32 seconds can affect the median task latency in addition to the variance. Figure 4.7 demonstrates that decreasing the threshold causes more workers to be replaced during a run, as expected. Figure 4.8 shows the latency percentiles at different worker-age slices (e.g., < 5 tasks) in the experiment. Varying the threshold affects both the median and higher percentiles, with a more pronounced effect on the extrema task latencies. For this workload, the optimal threshold is PM_8, which can reduce the straggler latencies by nearly 2×. However, further reducing the threshold to 4 or 2 seconds goes beyond the point where even fast workers are able to complete tasks, and effectively replaces all workers with the mean of the underlying MTurk distribution. The curves reduce across work slices due to the effects of pool maintenance, consistent with the analysis in Figure 4.5.

4.6.3 Straggler Mitigation

In this section, I evaluate the performance of straggler mitigation along two key metrics: task latency and task variance. An important parameter of straggler mitigation is R, the ratio of workers in the pool to tasks in a batch (Table 4.3), because it controls how many workers are assigned on average to eliminate stragglers. Set too low, and stragglers will occur unfettered. Set too high, and money and effort will be wasted unnecessarily. In these experiments, I set task size to $N_g = 5$, the pool size to $N_p = 15$, and give workers CIFAR-10 tasks.

Variance. One of the key properties of straggler mitigation is its ability to reduce the variance of individual task latencies. Figure 4.9 plots the standard deviation of the latencies of task completion times for each batch. Straggler mitigation consistently decreases the standard deviation by 5 to 10× (a decrease in variance of up to 100×!), very important when trying to predict the run-time of a batch consistently. One interesting observation is the jaggedness of the $R = 3$ plots. This is likely because with 3 times as many workers as tasks, workers spend much more time waiting, and are slow to respond when work becomes available because they are involved in other work.

Latency. Because straggler mitigation enables task batches to finish without waiting for high-latency straggler task assignments to complete, it significantly reduces the latency of each batch, up to 5× on some runs (Figure 4.10). Increasing R can increase those gains, but comes at an additional cost, as it pays more workers to complete each task. Although intuitively one might expect straggler mitigation to become more and more effective as R increases, there are practical limitations that prevent this effect. With high R, even fast workers are often terminated before finishing their tasks because many workers are working

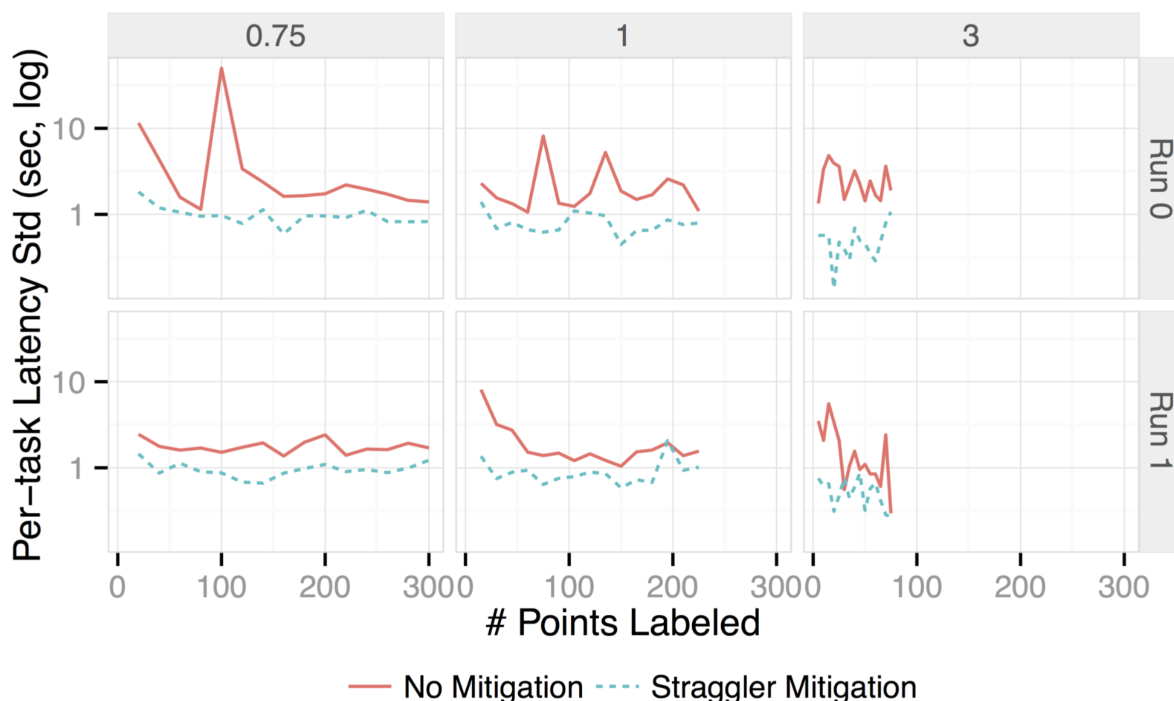


Figure 4.9: Straggler mitigation dramatically reduces the standard deviation of per-task latency across batches.

on every task at once. In addition to the added latency of this termination (workers must click a dialog to finish the old task and be presented with a new one, which takes seconds), this creates a frustrating environment for workers, who feel as though they aren't being allowed to work. As a result, keeping R between 0.75 and 1 is attractive, as it limits cost and still shows impressive speedups. The effects of straggler mitigation are summarized in figure 4.11.

4.6.4 Combining Per-Batch Techniques

Figure 4.12 summarizes the effects of combining both straggler mitigation and pool maintenance when labeling CIFAR-10 tasks. The two techniques can be complementary, but in some experiments we observe destructive interference between straggler mitigation and pool maintenance. This is most likely the result of fluctuating conditions on the underlying crowd platform across experiments: sometimes the initial pool selection is high-quality, rendering pool maintenance ineffective, and other times very slow workers join the pool and maintenance is invaluable. In all cases, it is worth noting that combining per-batch techniques still

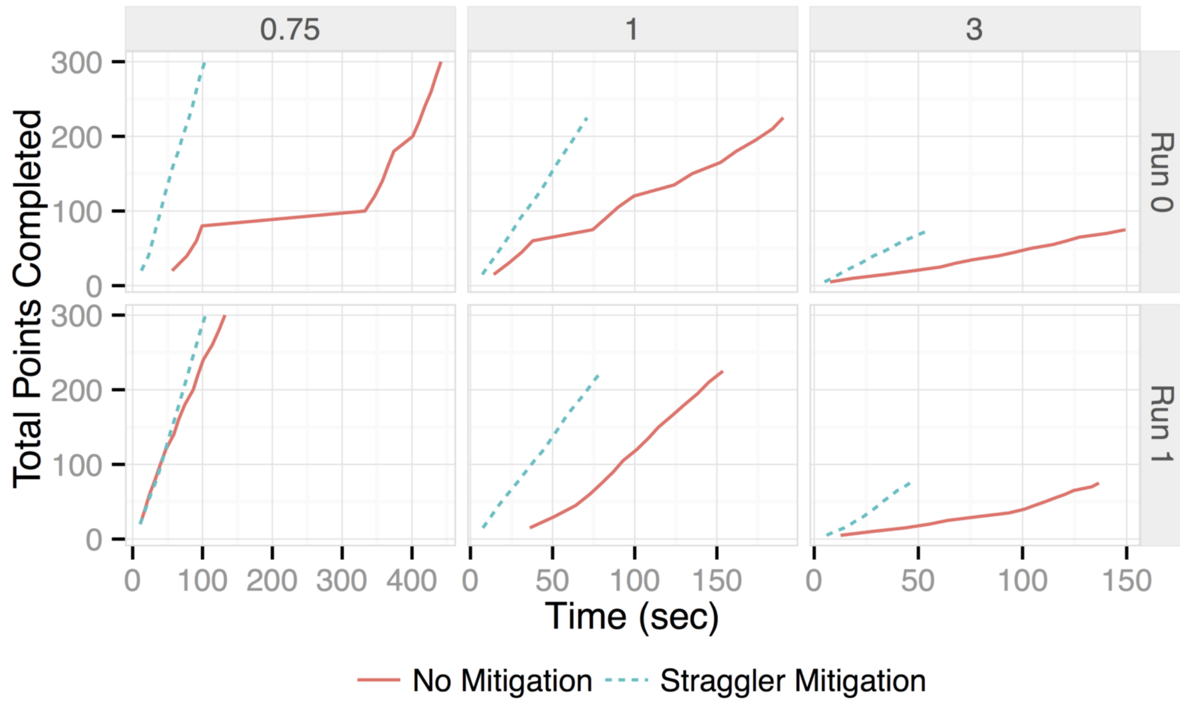


Figure 4.10: Points labeled over time with straggler mitigation in 2 typical runs.

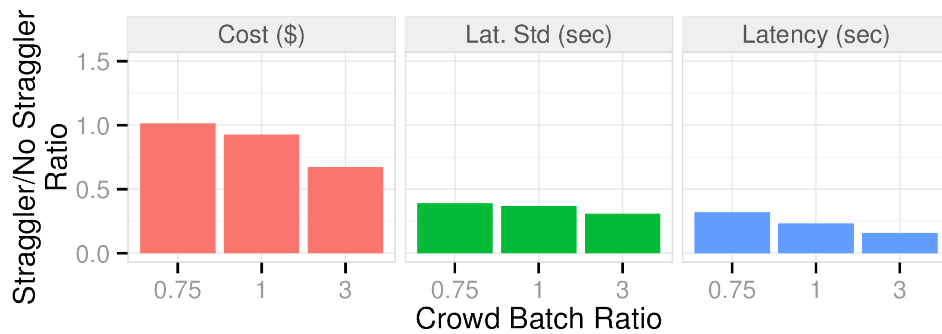


Figure 4.11: Straggler mitigation increases costs by 1 to 2×, improves latency by $2.5 - 5\times$, and variance by $4 - 14\times$.

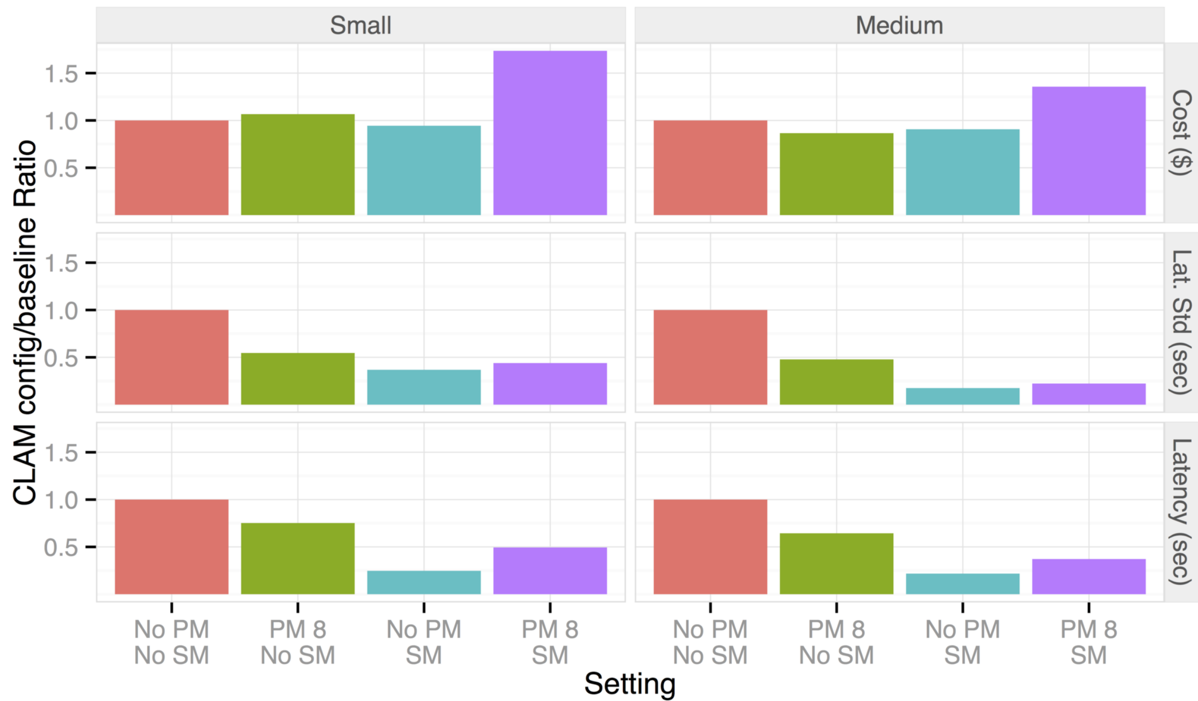


Figure 4.12: End-to-end Latency, Variance, and Costs for different straggler mitigation and pool maintenance configurations.

Figure 4.13: Per-assignment view of each straggler mitigation and churn configuration. Each horizontal segment is the length of an assignment. Red and blue dots denote batch boundaries

Figure 4.14: Replacement rate when using `TermEst` (Section 4.4.3) with $\alpha = 1$.

results in a significant speedup over not using either technique, leading to a reduction in latency of up to 6×, and reduction in standard deviation of up to 15×.

Detailed View. Figure 4.13 shows the latency of every task for a single experimental run with every combination of straggler mitigation and pool maintenance. Each line segment depicts the start and end time of a specific task. Red tasks are successfully completed, while blue tasks are terminated due to the worker leaving the pool or because another worker finished the task in less time. Red and blue dots denote the start and end of a batch, and the tasks completed by a given worker are aligned vertically along the y-axis.

The top two subplots show the value of pool maintenance – although stragglers are still present under pool maintenance, there are considerably fewer and lower magnitude stragglers as compared to the baseline pool. The bottom two subplots show that maintenance can further improve straggler mitigation by reducing the number of stragglers that must be ameliorated.

Effect of `TermEst`. Figure 4.14 measures the effectiveness of our model for estimating the latency of terminated tasks (Section 4.4.3). As expected, without `TermEst`, the worker replacement rate decreases dramatically, because workers are estimated to be faster than PM_ℓ and are not replaced. Adding `TermEst` adjusts for the gap: with it turned on, replacement happens just as frequently as with no straggler mitigation.

4.6.5 Hybrid Learning

In this section, I evaluate the hybrid learning strategy, demonstrating that it is effective on datasets where either active or passive learning would perform better, and that it successfully takes advantage of pool parallelism to reduce the time required to train a good model.

Accuracy. The Hybrid algorithm depends on the assumption that active learning does not outperform passive learning in all settings. Figure 4.15 validates this assumption in the

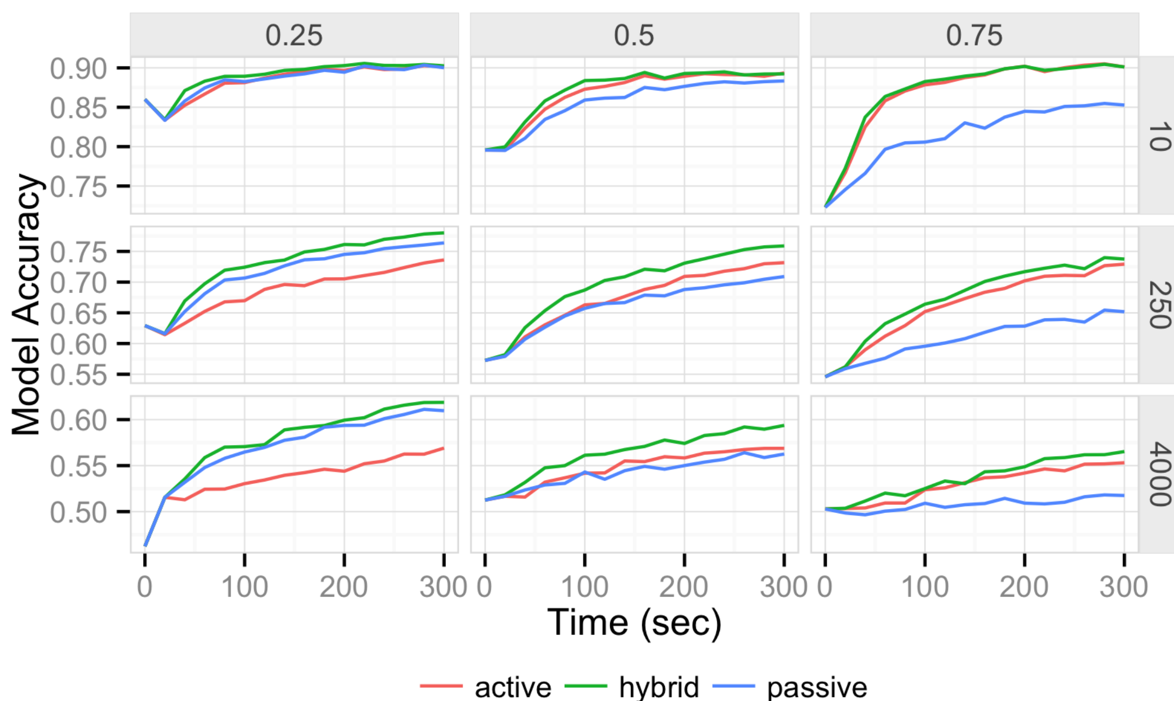


Figure 4.15: Active, Passive, and Hybrid strategies for learning on crowds run on generated datasets in the simulator.

simulator. It plots learning curves for active and passive learning on generated datasets of increasing hardness (rows show number of generated features), and shows how each learner performs given different amounts of the crowd's resources (columns show the percentage of the crowd pool used for active learning). On easier datasets, active learning significantly outperforms passive learning, but when given as many resources as active learning, passive learning is the better choice on harder learning tasks where active point selection is ineffective. This reinforces our belief that a successful hybrid strategy can trade off between the two approaches, and the hybrid lines in both Figure 4.15 and Figure 4.16 (wherein we replicate the simulator results on real-world datasets with live workers) demonstrate that the strategy is indeed successful. In all cases, hybrid performs as well as or better than either active or passive learning.

Latency savings. As a result of the fact that hybrid learning leverages the full parallelism of the crowd (as opposed to active learning with a limited batch size), the hybrid learning strategy is able to train better models faster. Figure 4.16 shows the hybrid learning strategy's performance compared to pure active or pure passive over time on the MNIST and CIFAR

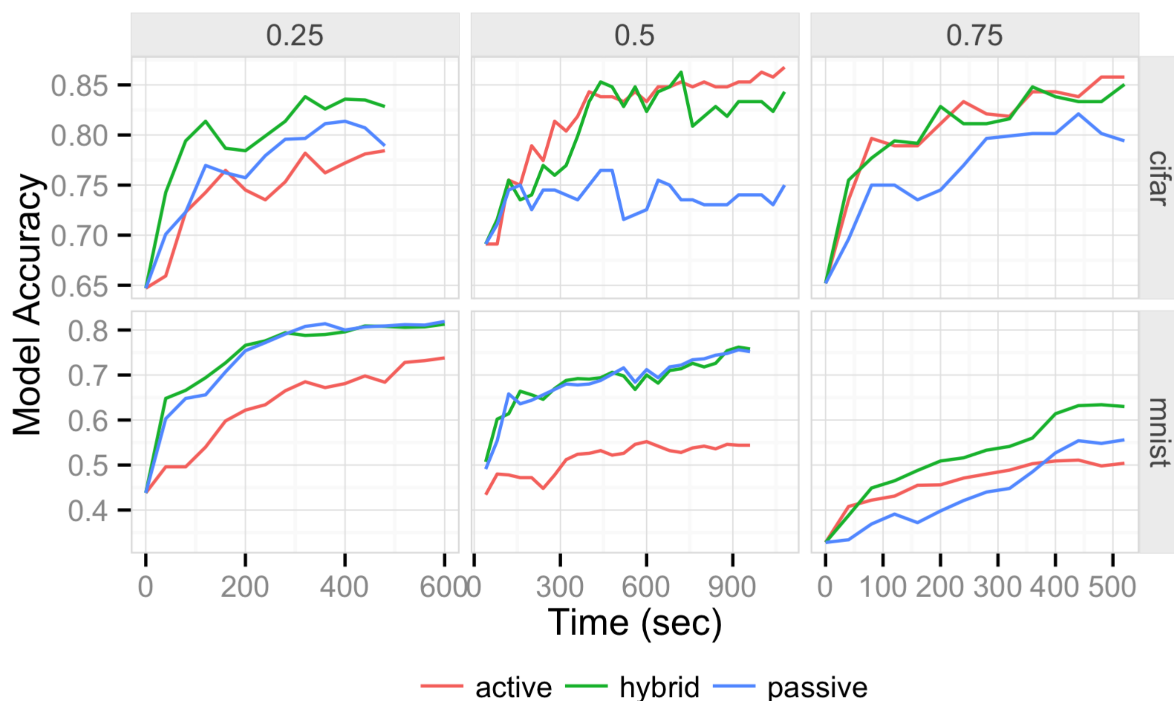


Figure 4.16: Active, Passive, and Hybrid strategies for learning on crowds run on real-world datasets on live workers.

datasets. The x and y axes of each plot show the accuracy improvement over time as points are labeled, the rows depict the datasets, and the columns represent the setting of the AL batch size as a percentage of the crowd pool size. In the same amount of time, the hybrid strategy is always the preferred solution for model training. In fact, on average, hybrid trains models of 85% accuracy on CIFAR (70% accuracy on MNIST) 1.2× (1.7×) faster than pure active learning and 1.6× (1.2×) faster than pure passive learning.

4.6.6 End-to-End Evaluation

In this section, I evaluate the end-to-end performance of CLAMShell against two baselines. `Base-NR`, which represents a typical crowd labeling deployment, sends labels out all at once, uses no retainer pool, and trains passive learning models to infer labels for unlabeled records. `Base-R`, which leverages the latest techniques for low-latency crowdsourcing, uses a retainer pool to label points in batches and active learning to infer labels for unlabeled records. In this experiment, 500 points were labeled by each strategy on the CIFAR-10 and MNIST datasets, and the accuracy of the resulting models were measured.

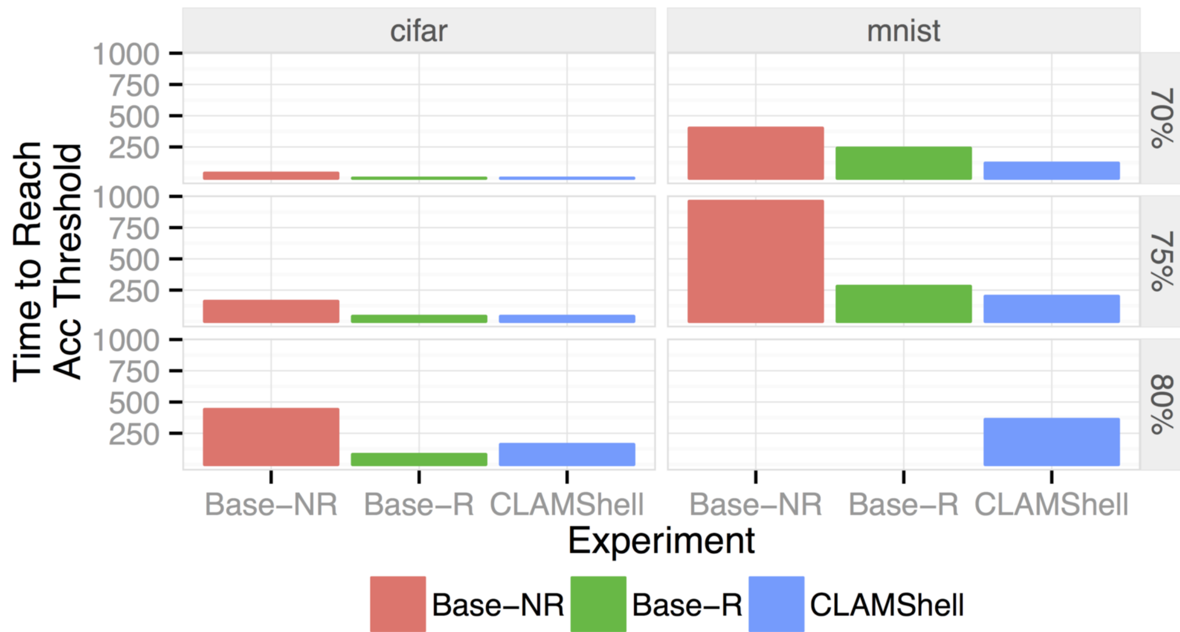


Figure 4.17: Summary of end to end time to reach models of varying accuracy

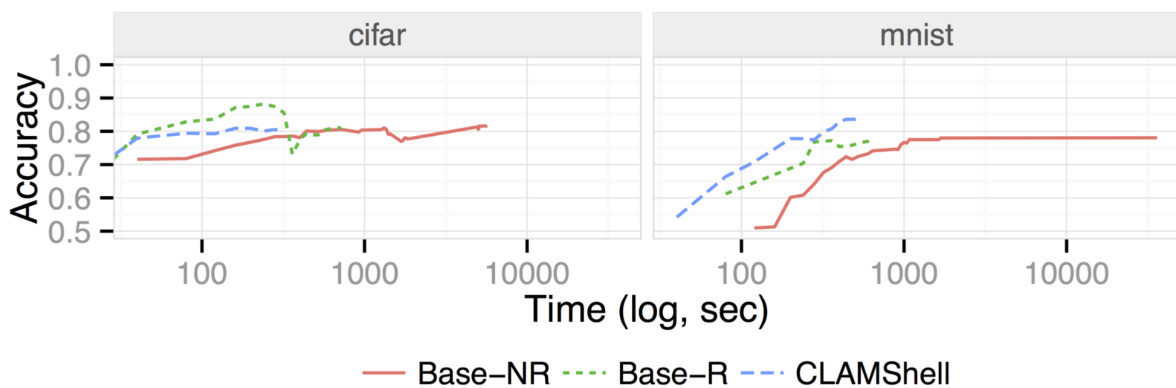


Figure 4.18: Wall clock time vs Model Accuracy

Results. Figures 4.17 and 4.18 summarize the results of this evaluation. In Figure 4.17, the rows represent an accuracy threshold for the model, and the plots show the wall-clock time taken by each strategy to train a model of that accuracy. Note that neither baselines reach an accuracy of 80% on the MNIST dataset in 500 points. To reach an accuracy of 75%, CLAMShell requires 4 to 5× less time than `Base-NR`. On the CIFAR dataset, CLAMShell does not significantly outperform `Base-R`, because passive learning performs well on CIFAR and the benefits of hybrid learning are limited. Figure 4.18 displays the full learning curves for each strategy, demonstrating that CLAMShell dominates both baselines in terms of model accuracy.

I also measured the raw time to acquire 500 labels from the crowd, and found that CLAMShell increases the labeling throughput by 7.24× compared to `Base-NR`. In addition, CLAMShell reduces the variance of labeling by 151×, and the absolute values are extremely low: 3.1 seconds vs. 475 seconds.

4.7 Conclusion

In this chapter, I have introduced CLAMShell, a system for data labelling that acquires labels from human crowd workers at interactive speeds. Latency can arise from many points in the labeling lifecycle, and CLAMShell addresses the key sources of latency with techniques that explore novel regions of the cost-accuracy-speed tradeoff. Straggler mitigation reduces the variance of task latencies within a batch by assigning additional workers to complete the task. Pool maintenance increases the average speed of workers in a labeling pool by replacing slow workers with faster ones over time. Hybrid learning reduces end-to-end labeling time by combining the fast convergence of active learning with the parallelism of passive learning.

All told, in our experiments CLAMShell reduces the latency of data labeling by an order of magnitude, and its variance by over 150×. These techniques make it possible to process large datasets with the crowd in just one or two minutes, timescales that are consistent with processing steps in automated data analysis pipelines. As a result, it is now possible to design crowd-powered components that fit into interactive systems for data analysis.

The techniques introduced in this chapter make two important assumptions: that the size of the worker pool is fixed, and that only a single user sends tasks to the crowd at a given time. In the next chapter, I relax both of these assumptions to provide support for a more general class of performant crowd-powered systems for data analysis. In doing so, I demonstrate opportunities for further performance improvements that arise when the size of the pool is adjusted to meet user workloads and workers are balanced efficiently between multiple users' requests.

Chapter 5

Cioppino: Multi-Tenant Crowd Management

The CLAMShell system presented in Chapter 4 demonstrated that a single user application running crowd tasks on a fixed-sized pool of crowd workers can be accelerated to near-machine latencies. However, the human computation needs of organizations frequently span multiple use cases, and it would be inefficient to implement individual crowd-powered systems to support each application. A better idea is to follow the approach taken by crowd platforms: maintain a single group of workers and assign them to work on the tasks requested by multiple applications. Because workers are shared between applications, the system can make global decisions about which tasks to assign to which workers, and react to changes in the volume of requests issued by each application.

In this chapter, I extend CLAMShell with support for multiple user applications running at the same time. The chief challenge to doing so is managing the workforce: sizing it correctly to support the workloads of the system's many users, maintaining that size as workers leave the system or workloads change, and assigning workers to applications on whose tasks they actually prefer to work. While CLAMShell assumed that the crowd pool size was fixed and determined by operational constraints, here I relax that assumption, and demonstrate that allowing the system to auto-scale the pool to support incoming requests can yield significant performance improvements over maintaining static pool sizes.

More so than in the previous chapters, this chapter focuses on the human aspects of crowd workers, which can all too easily be lost among systems analogies and abstracted mathematical theory. In general, being explicit about which aspects of human behavior (incredibly complex and impossible to model in its entirety) a system accounts for, how those aspects are modeled, and how the system has been designed to react to them is an important step in the design of any crowd-powered system. The work described here very intentionally models a set of human behaviors and preferences that can affect system performance and worker satisfaction in a multi-user system, and designs techniques that account for them directly.

5.1 Introduction

Most work in low-latency crowd systems, including that presented so far in this thesis, has assumed that tasks performed by human workers are short and homogeneous. However, at organizations that rely heavily on human computation for data analysis, workers often work on a variety of tasks of varying difficulty and length [100]. These tasks range from simple binary question answering, such as labeling training data for machine learning models [104, 127, 33] or identifying duplicate records for entity resolution [58, 142], to complex and content-heavy work such as data extraction [63], text translation [23] or video analysis [31].

A system where workers work on heterogeneous tasks requested for multiple different use cases has been called a *multi-tenant crowd system* [49]. Optimizing the performance of a such a system presents unique challenges. The following example illustrates the issues faced by a multi-tenant crowd.

Example 5.1 *Company X maintains a searchable database of products with public APIs, and has hired a pool of workers on a popular crowd platform to improve the quality of their data and services. Some workers monitor and filter incoming data containing explicit content, others extract structured records from newly scraped data sources, and still others provide search relevance feedback to improve API results. Though Company X currently hires and trains workers separately for each of these tasks, they often find that spikes and lulls in the volume of work leave workers overloaded or idle.*

In this chapter, I address the challenges of Example 5.1 head on. How should Company X size their crowd to meet demand? Can the crowd be resized adaptively? Should workers be trained for just one type of work, or moved between task types to meet demand despite the training overhead? Following existing work on real-time crowdsourcing [17], I model a crowd pool as a queueing system where crowd workers process incoming tasks as they arrive. An important insight is that this queueing system bears many similarities to a cloud-based software-as-a-service (SaaS) provider. SaaS systems handle requests to multiple service endpoints while attempting to maintain high overall throughput at low cost. To do so, they leverage techniques such as elastic cluster resizing [5], handling of nodes that fail or leave a cluster [43], and resource sharing between applications that run on the same cluster [56]. A multi-tenant crowd system faces many analogous challenges: sizing the worker pool for the task workload, reacting when workers abandon the system, and ensuring that workers are assigned tasks that they enjoy. As a result, existing approaches to cloud service management provide insight into managing crowds.

However, because workers are human beings, they cannot be treated as idealized processors. Unlike CPUs, human crowd workers abandon tasks due to fatigue, confusion, or boredom. Individual workers exhibit varying preferences and skill for different tasks, and improve over time as they become familiar with their work. When modeling a crowdsourced computation system, these human factors simply cannot be ignored.

In this work, I explicitly model a subset of human factors which matter for system performance, and simulate multi-tenant crowd workloads to demonstrate the advantages of the model. I build on top of the CLAMShell system presented in Chapter 4 to extend performant crowd systems to the multi-tenant setting where pool sizes are not fixed. The state-of-the-art in low-latency crowd systems does not automatically adapt the workforce size to workload changes, responds to worker abandonment reactively as workers leave, and either ignores under-utilized workers or assumes they have a static set of skills and cannot be trained on new tasks. In contrast, this chapter describes Cioppino[1], a system that explicitly models worker abandonment, task preferences, and training overheads, and addresses them with three novel techniques. *Pool elasticity* dynamically resizes the pool of workers to meet application demand without over-recruiting, leveraging techniques from the cloud autoscaling literature. *Pool stability* models the problem of worker abandonment. It maintains a worker pool of constant size by automatically recruiting new workers in a fashion adaptive to the rate of worker departure. *Pool balance* shares idle workers among applications to maximize throughput, balancing application demands against worker preferences. Together, these techniques demonstrate that adjusting simple models to account for human behavior can significantly improve the performance of crowd systems. I evaluate Cioppino in simulation using both synthetic data and a trace from a live crowd pool deployment and show a 19× decrease in cost over the crowd worker management used in today's systems, while doubling worker preference for the tasks they are assigned and improving throughput by 20%.

5.2 Related Work

As described in Chapter 2, existing research has focused on optimizing crowd systems in three ways. Quality optimizations improve the accuracy of data processing tasks (e.g., by weighting worker responses based on inferred measures of worker skill [75, 83]). Cost optimizations improve the cost-efficiency of specific crowd-powered operators such as joins [101], filters [109], entity resolution [144, 58] or active learning [104, 133]. Latency optimizations have attempted to accelerate human processing steps by recruiting human workers faster or in advance [20, 16], adaptively changing task pricing to encourage participation [55, 24], modifying task interfaces to reduce individual worker latencies [86, 99], recruiting and releasing workers to converge on a crowd of faster, more accurate workers [118], or leveraging redundancy to mitigate the effects of straggling workers (Chapter 4 of this thesis). While these techniques indeed accelerate the rate at which crowds process data, they are mostly focused on the batch-oriented, single-application scenario in which a dataset is processed in a single task interface by workers already trained in using that interface. This ignores important realities of today's crowdsourcing systems: work is varied in its complexity and latency requirements, and workers have innate preferences and skills that make them better suited for some tasks than others.

[1] A delicious Italian stew of CLAMs and other seafood.

Technique	System Dynamic Addressed	Worker Pool Actions Taken
Pool Elasticity (5.4)	Changing task workload	Worker recruitment and release
Pool Stability (5.5)	Worker abandonment	Worker recruitment
Pool Balance (5.6)	Excess idle workers	Inter-pool worker transfer

Table 5.1: Summary of techniques used by Cioppino.

The practicalities of managing a crowd of workers completing tasks in real time has received little attention in the crowdsourcing systems literature. The closest work is that of Bernstein et. al. [17], which models a pool of workers using queueing theory and finds the optimal size for a pool to minimize wait times given task and worker arrival rates. The goal is to avoid 'misses', where a task is available for processing but all workers are busy. Also, the pool maintenance technique described in Chapter 4 and the algorithms described in Chapter 6 dynamically recruit and release workers from the pool to maximize the average speed of pool workers. The pool management techniques described in this chapter also aim to minimize pool and task latency, but additionally take into account variable task duration, worker preferences, and training overheads to accommodate the heterogeneous nature of shared crowd pools. Additionally, I introduce techniques for autoscaling the crowd pool to handle increased or decreased load adaptively.

Separate from system optimization, human factors that affect crowd worker performance have received significant attention. There has been research into the effects of worker expertise [119], fatigue [121], pool abandonment [17] and motivation [67, 120], along with attempts to model and address the issues. For example, Rzeszotarski et al. [122] train a machine learning model on observed worker behavior to predict task performance, and Ikeda et al. [71] design worker collaboration schemes that account for inter-worker affinity.

5.3 The Cioppino System

In this section, I describe the architecture and queuing model of Cioppino, as well as summarizing its novel techniques.

5.3.1 Cioppino: a multi-tenant crowd system

The goal of a multi-tenant crowd system is to process requests with low latency and high throughput for client applications with varying workloads and task types. Taking into account the design considerations of Section 2.4, Cioppino is domain-agnostic, uses retainer pools for high availability, is compatible with existing quality control strategies, and introduces several novel techniques (summarized in Table 5.1): autoscaling worker pools to improve latency and throughput, explicitly modeling and compensating for worker abandonment, and transferring workers between applications to maximize throughput with consideration for worker preference and training overhead.

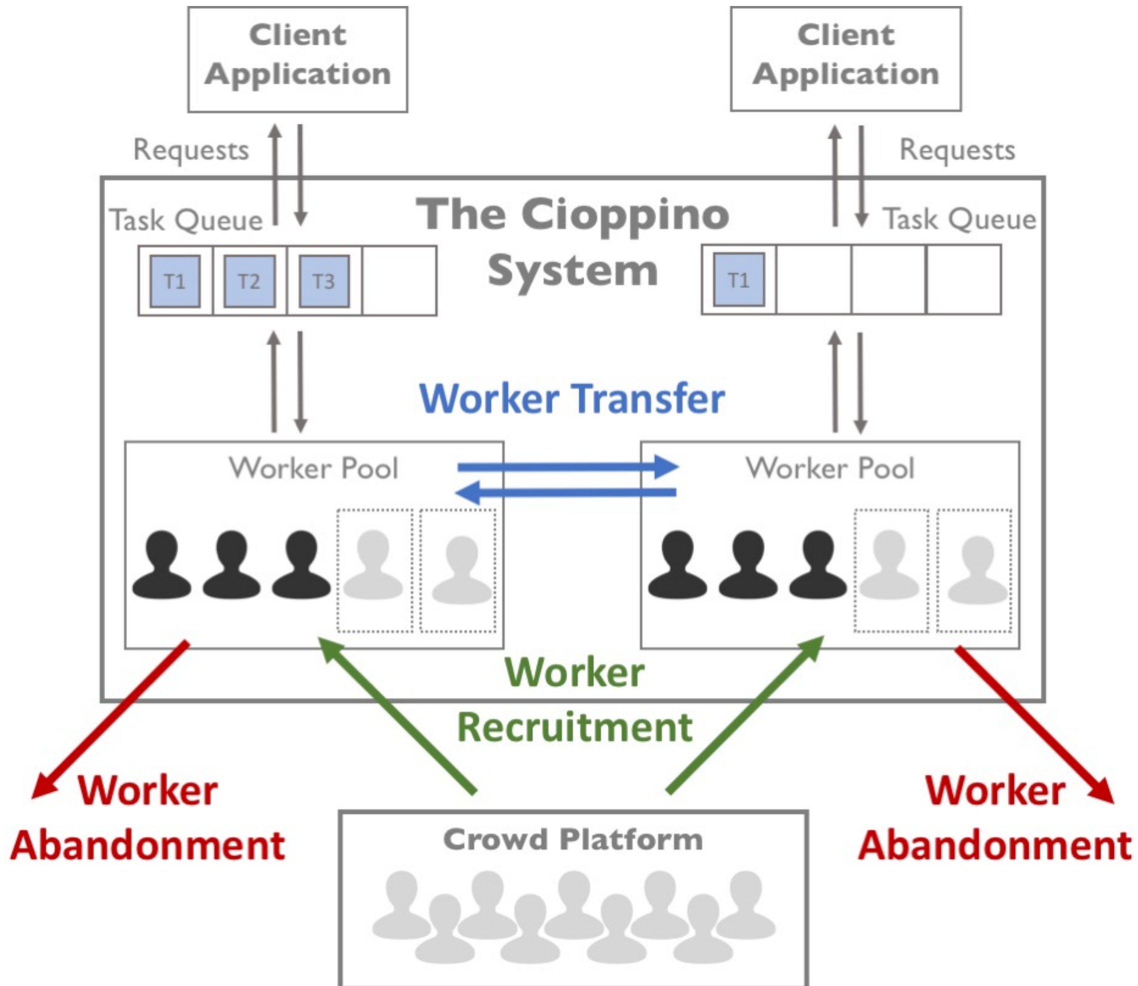


Figure 5.1: Cioppino architecture diagram.

A key insight is that managing crowds of human workers bears a strong resemblance to distributed cluster management. For example, cluster schedulers that support multiple applications must assign each task to a compute node, taking into account heterogeneous hardware and other factors that affect compute latency such as data placement [44, 140, 108]. Similarly, Cioppino assigns application tasks to human workers, accounting for varying worker preferences and training. Handling fluctuating task load by autoscaling compute clusters has also received attention. There are a number of approaches to autoscaling (see [116] for a recent survey). Techniques that translate well to the crowdsourcing setting include modeling the system with queues or queueing networks [5, 79] and making decisions to increase or decrease the cluster size using static rules [107], currently the state of the art for most cloud providers [6, 59], control theory [60], or machine learning [76].

Figure 5.1 shows the architecture of Cioppino. Client applications issue requests for tasks to be completed. Incoming tasks are enqueued and serviced by workers in a dedicated appli-

cation crowd worker pool. Cioppino recruits workers from the crowd to perform autoscaling (Section 5.4) and to compensate for worker abandonment (Section 5.5). Workers are also transferred between applications' worker pools to improve efficiency (Section 5.6).

5.3.2 System model

At its core, Cioppino is a queuing system: each type of crowd task (or *client application*) maintains its own queue of tasks, which is serviced by a pool of c_i workers held on retainer. Bernstein et al. [17] model retainer pools as M/M/c/c queues, in which workers arrive according to a Poisson process and are sent away if there is no demand for new workers. In this setup, however, queues hold arriving tasks, not arriving workers. Since it is undesirable to discard tasks, the system uses M/M/c queues (the theory of which is well-established: see [57] for an overview). In an M/M/c queue, the arrival rate of tasks is assumed to follow an exponential distribution with parameter λ_i, task completion times are distributed exponentially with parameter μ_i, and incoming tasks queue up (potentially infinitely) if there are no available workers. In practice, since the true values of λ_i and μ_i are unknown, the system periodically re-estimates them empirically from the observed task arrival and completion rates in a recent time window.

The value of the queuing model is that it permits analysis of the expected wait time for a task in the queue. With this information, one can estimate the optimal pool size to service a given workload (Section 5.4) and estimate the latency impact of transferring workers between applications' worker pools (Section 5.6). Now, I describe in turn each of the techniques Cioppino uses to manage crowd workers efficiently.

5.4 Pool Elasticity

In order to optimize its performance, Cioppino must determine a good size for the crowd pool, and adjust that size to meet changes in application workloads. Sizing the worker pool is an important decision. Too small, and its workers will be unable to keep up with the workload, causing a decrease in throughput. Too large, and workers will sit idle, incurring greater cost without any performance benefits. Cioppino leverages its core queueing theory model to identify an optimal queue size, and uses several algorithms for cloud autoscaling to adapt to changing workloads.

5.4.1 Optimal queue size

In the M/M/c queueing model, the tradeoff between performance and cost is captured as follows: the larger the number of workers c is, the lower the expected wait time for an incoming task (leading to a system-wide decrease in latency and increase in throughput). However, the larger c is, the higher the probability of idle workers. Under the retainer pool

model, the system must pay workers an additional salary to wait if there is no available work, so idle workers have a direct cost.

This tradeoff can be quantified as follows. Let $\rho = \frac{\lambda}{\mu c}$ be a measure of the 'traffic intensity' of the system. In an M/M/c queue, the probability that an incoming task must wait is given by

$$\Pi_W = \frac{(c\rho)^c}{c!}\left((1-\rho)\sum_{n=0}^{c-1}\frac{(c\rho)^n}{n!} + \frac{(c\rho)^c}{c!}\right)^{-1}.$$

This is referred to as Erlang's C formula, and can be used to compute both the expected wait time for incoming tasks,

$$\mathbb{E}[W] = \Pi_W \frac{1}{(1-\rho)c\mu} \tag{5.1}$$

and the expected queue length,

$$\mathbb{E}[L] = \Pi_W \frac{\rho}{1-\rho}.$$

The expected cost of the system's execution is given by the expected number of idle workers times the salary paid for waiting. Following [66], the salary is set to be $s = \$0.05$/minute, but this can easily be adjusted. So it is clear that

$$\mathbb{E}[Cost] = s(\max\{0, c - \mathbb{E}[L]\}). \tag{5.2}$$

To find an optimal pool size, Equations 5.1 and 5.2 must be explicitly traded off. This can be done by minimizing a weighted average of the two: $c^* = \arg\min_c \eta\mathbb{E}[W] + (1-\eta)\mathbb{E}[Cost]$, where η is a system parameter establishing a preference for one or the other. This equation has no clean closed form, but since it is convex, the exact optimum c^* can be solved for numerically.

5.4.2 Autoscaling algorithms

The optimal queue size c_i^* for an application derived in Section 5.4.1 depends on the parameters λ_i and μ_i. While μ_i is expected to remain fairly stable over time for a specific task type, λ_i varies with the application workload, which is not guaranteed to be constant. Cioppino intentionally estimates λ_i over a recent window of time to take this into account, and every time λ_i changes, the pool must be re-sized to compensate. Inspired by the literature on cloud infrastructure, Cioppino makes use of two types of autoscaling: rule-based autoscaling and control-theoretic autoscaling.

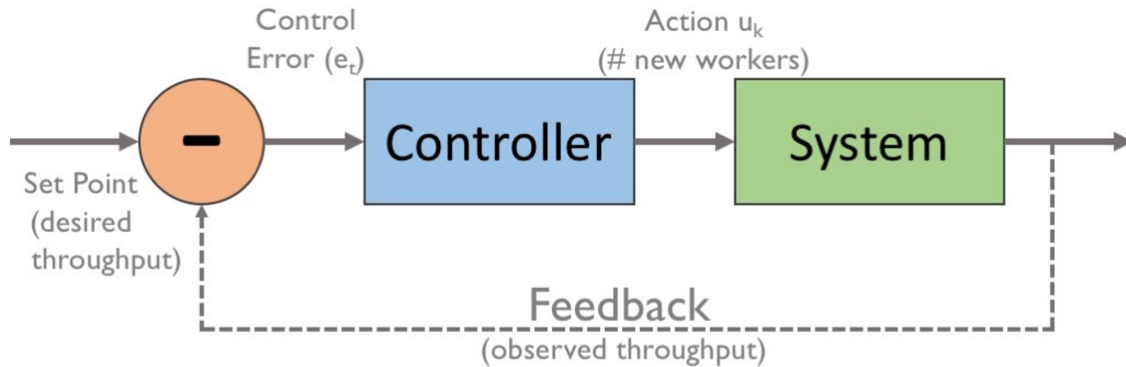


Figure 5.2: Control system block diagram.

Rule-based autoscaling. Most cloud IaaS providers (e.g., EC2 [6], GCE [59], etc.) currently offer threshold-based policies for autoscaling application hardware, for example, *"add 2 VMs to my application if the average CPU utilization among current VMs rises above 80%"*. This approach is attractive because it is easy to deploy and understand, but writing effective rules requires deep understanding of the application workload. In Cioppino, rules may be written based on system performance metrics (e.g., queue length, observed throughput, observed latency, pool size) or queueing model metrics (e.g., λ_i, μ_i, or the optimal pool size estimate c_i^*).

All rules are composed of a *condition* that triggers the rule and an *action* to take if the rule is triggered. A condition relates a parameter to a threshold, and may contain multiple clauses connected by boolean operators. For example, the following rule recruits a new worker whenever the queue builds up or the pool becomes smaller than the estimated optimum:

$$\texttt{if } L_i > 10 \texttt{ or } c_i^* - c_i > 5$$
$$\texttt{then action("recruit", 1).}$$

Rules that rely on c_i^* have particular promise, since they retain the ease-of-use of metric-based rules, but rely on the system model instead of requiring administrators to understand application workload.

Control-theoretic autoscaling. Systems for cloud autoscaling use a variety of control-theoretic models to compute the number of additional VMs that need provisioning, including fixed-gain controllers, adaptive controllers, and fuzzy models [96]. Cioppino uses fixed-gain PID controllers, which use a simple linear model to relate the control action to the observed error in the *process variable* or parameter of interest. Figure 5.2 shows a block diagram of our closed-loop control system. At each iteration of the loop, the user provides a target value (set point) for the process variable. The controller computes the desired change in the

number of crowd workers, then takes action to recruit or release workers. In a PID controller, the desired change in the number of workers is computed as

$$u_t = K_p e_t + K_i \sum_{j=0}^{t} e_j + K_d(e_t - e_{t-1}).$$

The first (Proportional) term reflects the current gap between the set point and the observed value of the process variable (e_t). The second (Integral) term reflects the history of observed errors. The third (Derivative) term reflects the current rate of change in error. The relative magnitude of these terms are controlled by the three weight parameters K_p, K_i, and K_d. We tune these parameters manually, though techniques for offline and online autotuning of PID systems has received considerable attention in the literature. In our experiments, we evaluate PID controllers built on two process variables: system throughput, for which e_t is the number of arrived tasks minus the number of completed tasks at timestep t; and pool size, for which e_t is the optimum c_i^* minus the pool size c_i at timestep t.

5.4.3 Cooling period

Though recruiting additional workers to a crowd pool is easily accomplished via platform API calls, it does not occur instantly. Rather, there is a non-deterministic delay until a new worker joins the pool. As a result, running an auto-scaling policy in a continual loop could lead to thrashing, where the system recruits and releases workers in rapid succession around an unstable equilibrium pool size. To alleviate this problem, I introduce a cooling period after each autoscaling decision wherein no additional autoscaling can occur. The length of the cooling period must be long enough to avoid thrashing, but not so long that the system cannot react to workload changes. A natural length is the mean worker recruitment time, so that on average additional autoscaling decisions will not be made until newly recruited workers have arrived. However, since worker recruitment times have high variance, Cioppino allows the cooling time to be set to k standard deviations above the mean, a parameter that can be adjusted upwards to reduce the odds of thrashing.

One additional consideration is that scaling down the crowd pool occurs instantaneously– the pool simply informs the worker that there is no work available to them, and stops sending tasks. In this case, Cioppino sets the cooling time to be the average task duration $\frac{1}{\mu_i}$, a period long enough that there will likely be observable changes in the metrics used to make scaling decisions.

5.5 Pool Stability

There are a number of challenges to running crowd retainer pools in production systems. One crucial limitation is that of worker abandonment.

5.5.1 Worker abandonment

In an idealized model of retainer pools, the requester hires a pool of workers, employs them until the desired work is complete, then pays and releases them. In reality, especially in online crowd marketplaces, workers abandon tasks without warning for a variety of reasons—fatigue, boredom, accidentally closing browser tabs, etc. For example, [16] found that about 10-20% of workers abandoned the pool before accepting their next task, and analysis of the CLAMShell trace described in Section 5.7.1 shows that workers on average only remain in the pool for 5-6 minutes. As a result, a production deployment of crowd worker pools must account for abandonment by constantly recruiting new workers to the pool. Cioppino introduces a novel mechanism to do so called *pool stability*.

5.5.2 Pool stability

Though it would be nice to be able to simply recruit another worker every time one leaves, the delay associated with hiring new workers might leave the pool short-handed for a significant period of time. Here, I examine three recruitment strategies to mitigate this effect: rule-based, average-rate, and hybrid. These approaches are evaluated in Section 5.7.3.

Rule-based recruitment. As with autoscaling techniques, rules can be explicitly defined to respond to worker abandonment. Cioppino permits rules of the form:

`if` $c_i < thresh$`, then action("recruit", n).`

Rules can also be defined in terms of the rate of abandonment. As before, rules are easy to define and apply, but hard to tune without knowledge of the dynamics of abandonment.

Average-rate recruitment. Rule-based approaches might perform poorly if the rule condition is poorly tuned to the abandonment rate. Average-rate recruitment seeks to avoid this problem. It continually maintains a count n_{window} of the number of workers who abandoned the pool in the last t_{window} seconds, then recruits $\hat{a}_i = \frac{n_{window}}{t_{window}}$ workers every second. If $\hat{a}_i$ is non-integral, then the algorithm recruits $\lfloor \frac{n_{window}}{t_{window}} \rfloor$ workers, and with probability $\frac{n_{window} \mod t_{window}}{t_{window}}$ recruits an additional worker. In expectation, this achieves the correct rate.

Hybrid recruitment. Average-rate recruitment avoids overreacting to spikes in abandonment by recruiting at a constant rate based on the empirical observed average. However, it ignores the current size of the pool, and cannot react if the pool size grows too large due to over-recruitment. Hybrid recruitment seeks to achieve the best of both worlds. It runs the average-rate recruitment algorithm, but additionally uses a threshold-based rule to prevent

recruitment if the pool is too large. That is, it can be summarized with the rule:

$$\texttt{if } c_i < thresh\texttt{, then action("recruit",}\\ \texttt{avg_rate_recruitment(}a_{window}, n_{window}\texttt{)).}$$

5.5.3 Interaction with pool elasticity

Since both pool stability and pool elasticity recruit workers automatically, the system must ensure that there is no destructive interference between the two techniques. To do so, all recruitment API calls are batched into short windows of time, and the net total of new workers is requested. For example, if pool elasticity attempts to release 2 workers (due to lightened workload) but pool stability attempts to recruit 1 worker (due to increased observed worker abandonment), the system will release 1 worker. This can be justified intuitively: pool stability recruits additional workers so that the pool behaves according to the queueing model that pool elasticity relies on. To avoid impacting performance, the length of the window should be made small enough so that two recruitment decisions are not made by the same pool within the same window. Due to the cooling period discussed in Section 5.4.3, this is not difficult: the experiments in Section 5.7 use a window of 1 second, and do not observe any impact on performance. One final point: since c_i might be changing due to both worker abandonment and autoscaling, rule-based techniques should be designed jointly to avoid redundant recruitment.

5.6 Pool Balance

Autoscaling each application's pool individually will successfully adjust to changing workloads, but it may result in inefficiency because it doesn't take into account the scaling needs of other pools. Cioppino uses a novel algorithm called *Pool Balance* to determine when it is efficient to shift workers between applications, taking into account two subtleties: first, there is a training overhead associated with assigning a worker a task type they've never done before; and second, worker preferences should be taken into account when performing transfers.

5.6.1 Approach overview

Algorithm 5.1 describes Cioppino's algorithm for choosing a set of idle workers to transfer to other pools (a *transfer set*, consisting of zero or more `(worker, destination-pool)` pairs). For each transfer set we consider, we simulate the transfers, then estimate how it will affect the processing of currently backed up tasks and the preferences of workers for their current work. Because the space of possible transfer sets is large, we use heuristics to limit the candidate transfer sets we consider. The pool balance algorithm runs in a loop during system execution: periodically, it generates candidate transfer sets, evaluates them, and applies the best one.

Algorithm 5.1 Pool Balance.

```
best_tset_score = −∞, best_tset = None
for tset in generate_candidate_tsets():
    completion_times = {}
    preferences = {}
    for pool in All_Pools:
        new_pool = simulate_transfer(tset, pool)
        completion_times += completion_est(tset, new_pool)
        preferences += net_preference(new_pool)
    perf_score = perf_ts(completion_times)
    h_score = h_ts(preferences)
    net_score = ω∗ h_score + (1 − ω)∗ perf_score
    if net_score > best_tset_score:
        best_tset = transfer_set
        best_tset_score = net_score
apply_tset(best_tset)
```

5.6.2 Estimating queue completion time

The function `completion_est` in Algorithm 5.1 estimates the time it will take for an individual queue to empty given its current length L_i and its current workforce c_i. This estimate is equivalent to the expected wait time of the next item to arrive in the queue, $\mathbb{E}[W_i|L_i]$. To compute the estimated waiting time, it can be observed that the task will wait in line until L_i items have been processed and exited the queue. Let D_k be the kth interdeparture time since the arrival of the last task in the queue. Then $\mathbb{E}[W_i] = \mathbb{E}[\sum_{k=1}^{L_i} D_k]$. Clearly, the D_k are independently exponentially distributed with mean $\frac{1}{c_i\mu_i}$. The sum of n independent exponentially distributed random variables with identical mean μ is Erlang-distributed with shape parameter n and rate μ, so we have that:

$$\mathbb{E}[W_i] = \mathbb{E}\left[\sum_{k=1}^{L_i} D_k\right] = \mathbb{E}[Er(x; L_i, c_i\mu_i)] = \frac{L_i}{c_i\mu_i}. \quad (5.3)$$

When workers are transferred between pools, however, they must learn to do the new task type. Thus, Equation 5.3 must be adjusted to account for training time. To simplify the analysis, Cioppino makes the conservative assumption that new workers aren't added to the pool until all workers have completed training. A transfer thus breaks queue processing time into two phases: during training, the queue can't make use of new workers, but once training is done, it can. Let us assume that training times, like task times, are exponentially distributed with parameter tr_i, let k_i^{TS} be the number new workers in pool i under transfer set TS, and let c_i^{TS} be the total workers after TS is applied: $c_i + k_i^{TS} = c_i^{TS}$.

First, I analyze the expected length of the training period. Training ends when all workers have finished training, so (using that the expectation of the maximum of n independent

exponential random variables with parameter λ is $\frac{1}{\lambda}\sum_{j=1}^{n}\frac{1}{j}$ — see [97] for a proof):

$$\mathbb{E}[t_{train}] = \mathbb{E}[\max(tr_{i,1}, \ldots, tr_{i,k_i^{TS}})] = \frac{1}{tr_i}\sum_{j=1}^{k_i^{TS}}\frac{1}{j}.$$

During training, the queue processes tasks at an average rate of $\mu_i(c_i^{TS} - k_i^{TS})$, so an expected $\frac{\mu_i(c_i^{TS}-k_i^{TS})}{tr_i}\sum_{j=1}^{k_i^{TS}}\frac{1}{j}$ tasks exit the queue during training. Call this quantity L_{tr}. Now the adjusted $\mathbb{E}[W_i]$ can be expressed in terms of the training and post-training phases:

$$\mathbb{E}[W_i^{TS}] = \begin{cases} \frac{L_i}{(c_i^{TS}-k_i^{TS})\mu_i} & \text{if } L_i \leq L_{tr}, \\ \mathbb{E}[t_{train}] + \frac{L_i - L_{tr}}{c_i^{TS}\mu_i} & \text{otherwise.} \end{cases} \tag{5.4}$$

That is, if the queue is small enough to be completely processed during training, then the expected processing time is as if the new workers didn't exist. If not, then the expected processing time is the expected training time plus the time to process the tasks that weren't processed during training, using the newly trained workers.

5.6.3 Evaluating candidate transfer sets

Performance. With an estimate for the expected time to process an individual queue, the performance utility of a transfer set TS can be evaluated. First, the set $\mathcal{W} = \{\mathbb{E}[W_i^{TS}] : 0 \leq i < |Apps|\}$ is computed using Equation 5.4. Then, the performance score $Perf^{TS}$ (the `perf_ts` function in Algorithm 5.1) is computed by evaluating the utility of this set. The experiments in section 5.7 use $Perf^{TS} = 1/\text{median}(\mathcal{W})$, though other utility functions could be used based on system performance goals.

Worker preferences. $Perf^{TS}$ only accounts for the performance effects of a transfer set. However, transferring a worker may be undesirable if the worker doesn't enjoy the task type of the destination application, and two equally efficient transfer sets might not be equally sensitive to workers' task type preferences. To model this, let $p_{i,j} \in [0, 1]$ represent the jth worker's preference for the task type of application i. In simulation, the $p_{i,j}$ are generated uniformly at random, but in a live system new workers could establish preferences for the available task types when they join. To evaluate a transfer set TS, in addition to computing $\mathbb{E}[W_i^{TS}]$, the pool balance algorithm computes the *net preference* of the transfer set,

$$H^{TS} = \sum_{i \in Apps}\sum_{j \in Apps_i^{TS}} p_{i,j}. \tag{5.5}$$

Then, overall transfer set score is updated to take the weighted average of the two metrics:

$$Score^{TS} = \omega * H^{TS} + (1 - \omega) * Perf^{TS}, \tag{5.6}$$

where ω is a system parameter that sets the relative importance of worker satisfaction to system performance. I evaluate the pool balance algorithm's effect on net preference in Section 5.7.4.

5.6.4 Searching the transfer set space

Unfortunately, the space of candidate transfer sets is large, making a brute-force comparison of all possible transfer sets impractical. To see this, observe that any subset of idle workers in the system could be transferred to any other app in the system, so if there are A apps in the system and $C = \sum_{i=0}^{A-1} c_i$ total workers, there are up to $\sum_{j=0}^{C} \binom{C}{j} A^j = (A+1)^C$ possible transfer sets.

Instead, Cioppino applies heuristics to cut down the space it searches over. First, the system doesn't consider transfer sets that send workers to applications having $L_i = 0$, since those applications are unlikely to need additional workers. Note that this rules out transfer sets that swap workers symmetrically between pools, since all applications with idle workers have $L_i = 0$. Additionally, the size of a transfer set is capped at 10. Finally, instead of considering all A^k transfer sets for a worker subset of size k, the system uses a greedy heuristic to generate a single assignment of workers to destinations. The heuristic works as follows: for each worker pool, the system computes $\mathbb{E}[W_i]$ from Equation 5.3 and H^{TS} from Equation 5.5. Then each worker in the subset is assigned to the destination pool with the largest $\omega * H^{TS} + (1 - \omega) * \mathbb{E}[W_i]$, incrementing c_i and recomputing the score after each assignment.

Further, the system uses a caching approach to avoid redundant computation while searching the space. In order to evaluate candidate transfer sets, the pool balance algorithm repeatedly computes $\mathbb{E}[W_i^{TS}]$ according to Equation 5.4. Because this computation relies only on $c_i^{TS}, k_i^{TS}, \mu_i$, and L_i, results can be reused whenever simulating a transfer set results in an application with a previously seen $c_i^{TS}, k_i^{TS}, \mu_i$, and L_i. The algorithm maintains a hash table mapping the tuple (c_i, k_i, μ_i, L_i) to the result $\mathbb{E}[W_i]$, and uses it whenever possible. This occurs fairly often—for example, there will be many ways to transfer k workers to application i from different source applications, and for each, $\mathbb{E}[W_i^{TS}]$ will be identical. The map remains valid across multiple iterations of transfer as well, so it is maintained across runs of the algorithm.

5.7 Evaluation

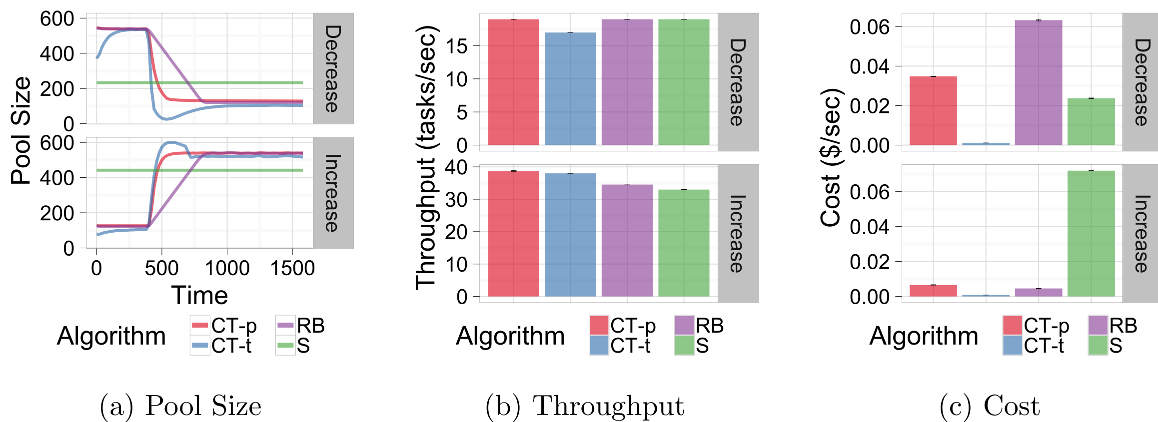
In this section, I evaluate Cioppino's techniques in isolation and together, demonstrating simultaneously a 20% improvement in throughput and a 19× decrease in cost over the current state of the art in crowd pool management.

5.7.1 Experimental Setup

I evaluated Cioppino's techniques on a multi-tenant crowd pool simulator implemented in python. The experiments were conducted on an Amazon EC2 c4.4xlarge instance with 16 vCPUs and 30 GB of RAM. The workloads and crowd behavioral data were either generated or trace-based.

Generated data. In some experiments, crowd worker behavior is simulated, including the rate of crowd worker abandonment, the recruitment time for new workers, and preferences of workers. Abandonment rates are modeled by a parameter α: each worker chooses to abandon the system with probability α at the end of their task. Recruitment time is modeled as a Poisson process with rate r. Worker preferences are generated uniformly at random in $[0, 1)$ for each task type (client application) in the system, then normalized to sum to 1. In addition, workload data are simulated, including the task completion rate μ_i, the arrival rate of tasks λ_i, and initial pool sizes c_i for client applications. Specific workloads will be described in the context of the experiments that use them.

Trace-based data. Some experiments are trace-based simulations. The trace was extracted from a live deployment of the CLAMShell system running simple labeling tasks on MTurk described in Chapter 4.[2] It consists of three streams of data in chronological order: the worker recruitment times for all new workers who joined the pool (n=6126 recruitments, mean time 91.3 seconds, standard deviation 113.9 seconds), the task duration for all tasks processed in the pool (n=6725 tasks, mean time 1.97 seconds, standard deviation 0.87 seconds), and the duration that workers remained in the pool before abandoning it (n=2999 abandonments, mean time 316.6 seconds, standard deviation 211.1 seconds).



(a) Pool Size (b) Throughput (c) Cost

Figure 5.3: Response of pool elasticity algorithms to workload decrease or increase, averaged over 50 runs. The workload change begins at the 400 second mark.

5.7.2 Pool elasticity

First, I evaluate the Pool Elasticity techniques that autoscale crowd pools in response to changing application workloads. I compare four techniques. Rule-based autoscaling (RB) uses two rules: "`if` $L_i > 10$ `or` $c_i^* - c_i > 5$ `then action('recruit', 1)`", and "`if`

[2]The trace data can be found at `http://thisisdhaas.com/clamshell_trace.html`.

Abbrev.	Technique	Description
RB	Rule-based	Scales pool when rule conditions are met.
CT-t	Control-theoretic (throughput)	PID controller with throughput as process variable.
CT-p	Control-theoretic (pool size)	PID controller with pool size as process variable.
S	Static	Constant pool size, set to the workload optimum.

Table 5.2: Summary of pool elasticity techniques evaluated in Section 5.7.2.

$c_i^* - c_i < -5$ `then action('release', 1)`." Control-theoretic autoscaling uses the PID controller described in Section 5.4.2, and is evaluated with two different process variables: system throughput (CT-t) and pool size (CT-p). Finally, the static technique (S) begins with the optimal pool size for the entire workload and does no autoscaling, for comparison. The static technique cannot be used in practice without advance knowledge of the entire workload. Table 5.2 summarizes these techniques.

To capture the responsiveness of the techniques to workload changes, I evaluate them against sudden increases and decreases in workload. In both cases, the optimal pool size is computed following the calculation in Section 5.4.1 with $\eta = 0.5$, which assigns equal weight to minimizing cost and maximizing performance. In the increase scenario, the workload starts at $\lambda_i = 10, \mu_i = 0.1$ (making $c_i^* = 126$), then increases to $\lambda_i = 50, \mu_i = 0.1, c_i^* = 545$. The decrease scenario reverses the order of the two phases. In both scenarios, the initial pool size c_i is set to the initial c_i^* so that the experiment begins at equilibrium. These experiments are run on synthetic, not trace-based data.

Figure 5.3a shows how the techniques react to workload changes. Qualitatively, though all techniques other than S converge to the new optimal value after a workload change, the control-theoretic techniques react more quickly. The technique based on the Cioppino queueing model (CT-p) most smoothly and rapidly converges to the new optimal value, where as CT-t has a tendency to overshoot.

Figure 5.3b evaluates the total system throughput achieved by the techniques. In the Decrease scenario, all techniques achieve high throughput, since throughput is unaffected if the crowd pool remains too large (though CT-t has a 10% lower throughput due to its overshoot, 17 tasks/sec vs. 19 task/sec for CT-p). In the Increase scenario, rapid scaling is rewarded, and the control theoretic techniques (CT-p=38.72, CT-t=38.0) see a 12% increase in throughput over RB and a 17% increase in throughput over S.

Finally, figure 5.3c shows the average cost per second of the techniques. Here the results are dramatic: CT-t costs 5-8× less than the other techniques and 88× less than the baseline

strategy S in the Increase scenario. In the Decrease scenario, the rule-based strategy is penalized for its slow reaction time (57× more expensive than CT-t), and the S and CT-p strategies are 21 and 31× more expensive as well. The dominance of CT-t arises from the fact that the PID control system is designed to keep throughput at exactly the same rate as task arrival, which when successful means there are no idle workers. In contrast, the setting $\eta = 0.5$ for CT-p means that the pool is intentionally kept slightly larger to trade off increased throughput for cost.

Summary: *If cost is a factor, CT-t is the obvious best choice for pool autoscaling. However, CT-p consistently achieves the best throughput by keeping the worker pool at the optimal size prescribed by the Cioppino queuing model.*

5.7.3 Pool stability

In this section, I evaluate the Pool stability algorithms that compensate for worker abandonment. In these experiments, a constant workload of $\lambda_i = 10, \mu_i = 0.1$ is maintained, and the initial pool size is set to be $c_i = c_i^* = 126$. This allows the impact of abandonment on normal performance under the null technique (N) to be measured. I additionally compare three techniques. Rule-based stability (RB) recruits new workers whenever the pool size dips too low, implementing the rule: "`if` $c_i^* - c_i > 5$, `then action('recruit', 1)`". Average-rate recruitment (AR) recruits new workers at a constant average rate. Hybrid recruitment (HR) uses the same rule as RB combined with the rate of AR, as described in Section 5.5.2. These experiments are run on synthetic, not trace-based data.

Figure 5.4 illustrates the performance of the various techniques under varying abandonment rates α and recruitment rates r. The baseline N illustrates the effect α has on how quickly workers leave the pool. As α increases, the rule-based technique (RB) starts losing more and more workers. Since rules have hard-coded thresholds, RB cannot adapt to changing alpha in the way that average-rate recruitment techniques can. The parameter r describes the responsiveness of the system to recruiting decisions: on average, recruited workers arrive after $\frac{1}{r}$ seconds. As a result, the pool size exhibits more variance as r decreases. Finally, the hybrid strategy (HR) acts as expected: it is a more conservative version of AR because it doesn't recruit if the pool size is sufficiently large.

Figures 5.5a and 5.5b demonstrate the techniques' average throughput and cost. Values of α are restricted to values observed in real deployments for readability. AR, the strategy that recruits most aggressively, shows the strongest performance, ranging from 5-15% higher throughput than HR, 14% - 2× higher throughput than RB, and up to 14× higher throughput than N. However, because it over-recruits, it incurs significant cost: 3.3× HR, and 47× RB.

Summary: *Though the AR strategy maintains the pool's stability and achieves high throughput, the HR strategy takes a small performance hit for significantly decreased costs. The RB strategy performs relatively well when the rule is tuned to a particular α, but does not react well to changes in the system dynamics.*

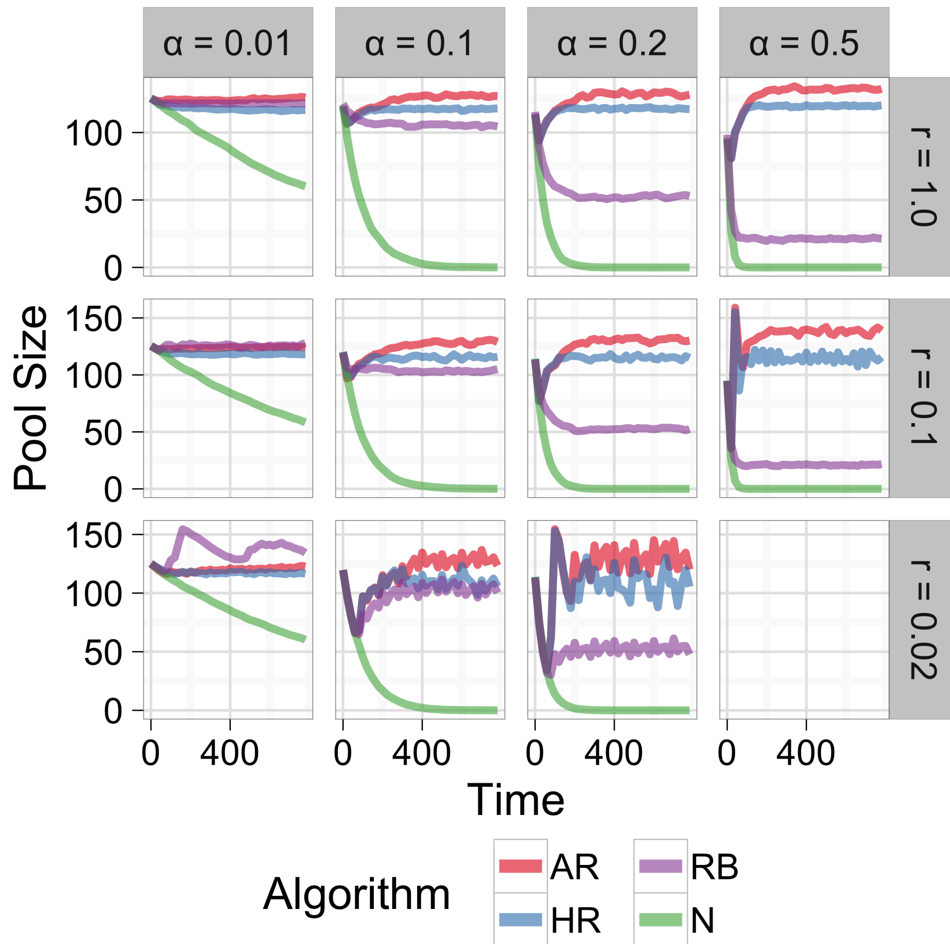
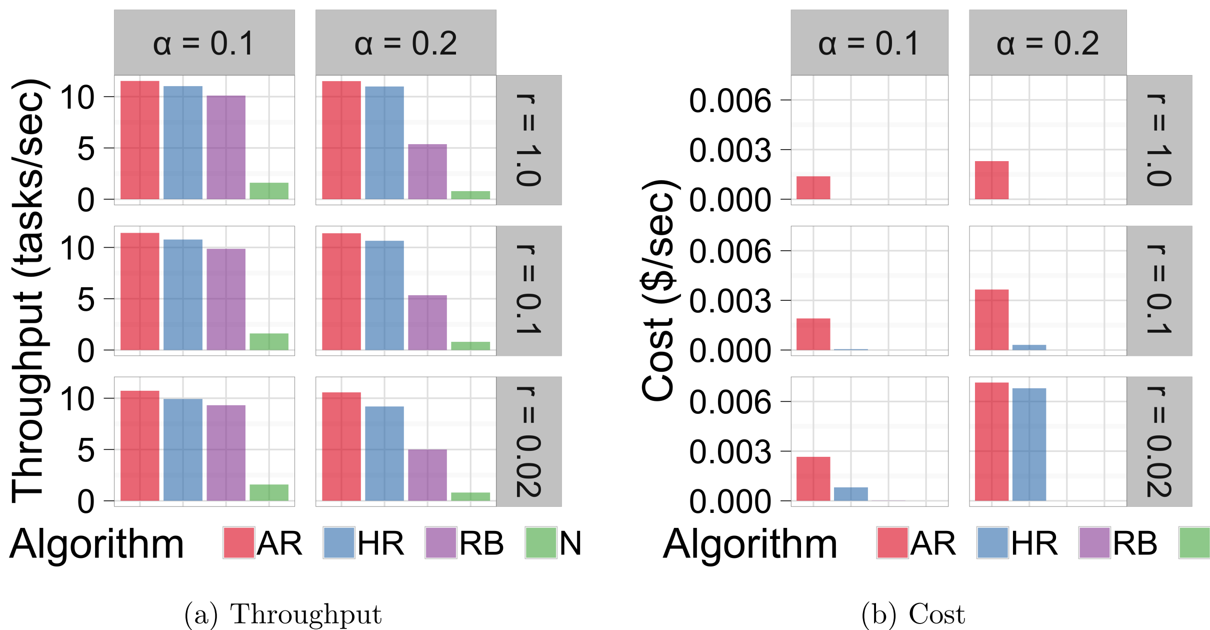


Figure 5.4: Pool stability techniques and the effects of the parameters α (the probability that each worker abandons the pool after completing a task) and r (the mean recruitment rate, $\frac{1}{r}$ is the average delay before a recruited worker arrives), averaged over 50 runs. The chart for $\alpha = 0.5, r = 0.02$ is omitted because the y axis would be scaled too high as a result of over-recruitment by AR and HR.



(a) Throughput

(b) Cost

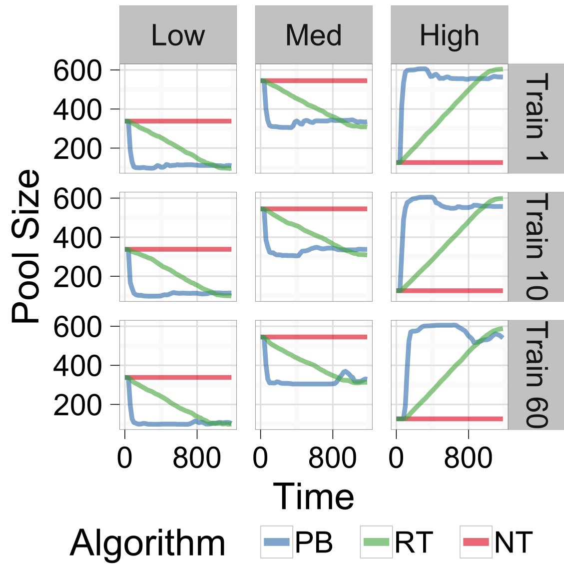
Figure 5.5: Throughput and cost of pool stability algorithms for varied α and r, averaged over 50 runs.

5.7.4 Pool balance

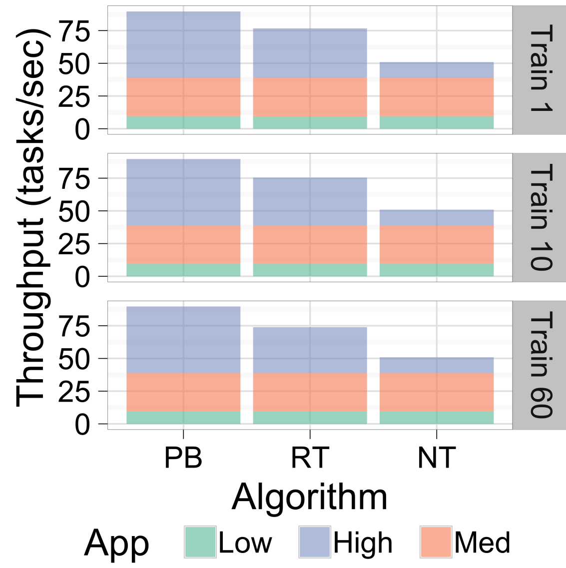
To evaluate the pool balance algorithm, I examine the net cost, performance, and worker preferences of using different strategies to transfer idle workers. In addition to the pool balance algorithm (PB) described in Section 5.6, I compare against two baselines: no transferring (NT), as is typically done in today's crowd deployments, and random transferring (RT), where all idle workers are immediately sent to a random pool with a queue length $L_i > 0$.

In order to observe how the algorithms rebalance workers, I run a multi-tenant pool with three applications: a high-load application ($\lambda_i = 50, \mu_i = 0.1, c_i^* = 545$), a medium-load application ($\lambda_i = 30, \mu_i = 0.1, c_i^* = 338$), and a low-load application ($\lambda_i = 10, \mu_i = 0.1, c_i^* = 126$). The system begins with the total optimal number of workers, balanced sub-optimally between the applications. The high-load application is given 126 workers, the medium-load application is given 545 workers, and the low-load application is given 338 workers. These experiments are run on synthetic, not trace-based data.

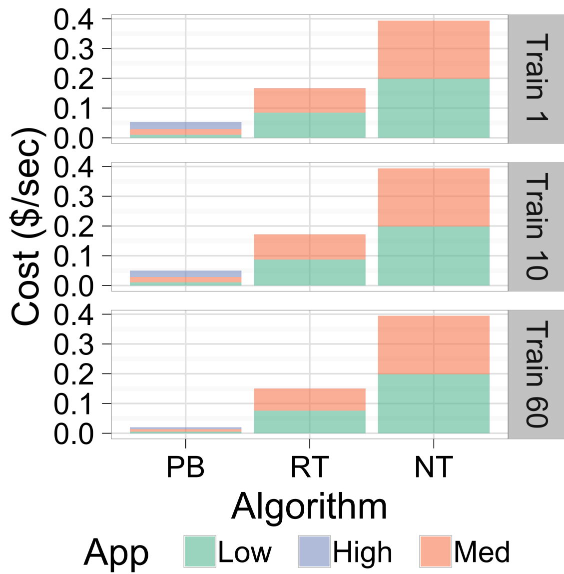
Figure 5.6a shows how workers are rebalanced across the applications by the various techniques. Both the RT and PB strategies successfully rebalance workers to arrive at the optimal pool size; however, the PB algorithm rebalances much more quickly, since it picks high-quality transfer sets. The speed of the rebalancing has important consequences for both performance and cost. Figure 5.6b shows that PB consistently achieves an average throughput about 18% higher than RT and 76% higher than NT. Figure 5.6c shows that PB



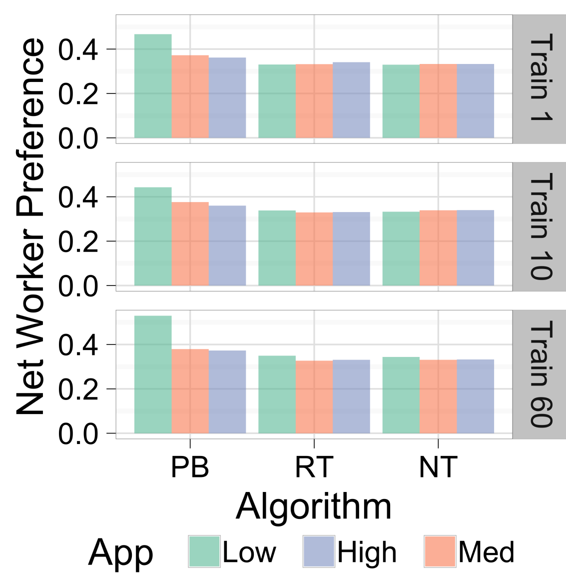
(a) Pool size



(b) Average throughput



(c) Average cost



(d) Net worker preference

Figure 5.6: Performance of pool balance algorithms on the 3-application system faceted by tr_i, averaged over 50 runs.

is also significantly less expensive, on average costing 10× less than RT and 27× less than NT.

Finally, Figure 5.6d shows the average preferences of workers in each application pool for their current task at the end of system execution (after workers have been rebalanced). While NT and RT show worker preferences of 33% per pool, equivalent to random assignment, PB is successful in making workers happier on average: up to 47% preference of workers for the pool they are assigned to. This might still sound low, but it represents a 43% increase in the average preference a worker has for the tasks they are assigned to work on.

Summary: *Cioppino's pool balance algorithm shows significantly higher throughput, lower cost, and increased net worker preference than the baselines.*

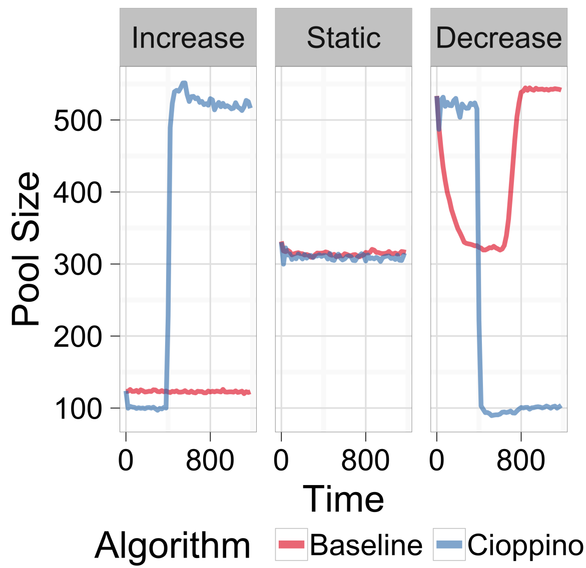
5.7.5 End-to-end evaluation

To evaluate the overall efficiency of Cioppino, I compare two scenarios, a state-of-the-art baseline and a combination of the best performing techniques of Cioppino. The baseline behaves like typical crowd systems today: it does not autoscale its crowd pool (pool elasticity policy S), it uses rule-based recruitment to respond to worker abandonment (pool stability policy RB), and it does not do any inter-pool worker training and transfer (pool balance policy NT). The Cioppino system uses throughput-based control-theoretic autoscaling (CT-t), hybrid recruitment for pool stability (HR), and the novel pool balancing algorithm (PB).

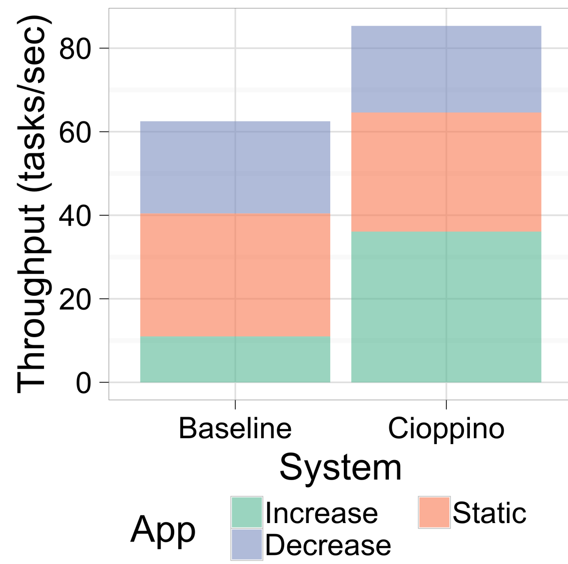
Synthetic data. In order to evaluate all techniques simultaneously, I use a synthetic workload designed to exercise each of Cioppino's crowd management techniques. It runs three apps simultaneously with different workloads. The Increase and Decrease apps are the same as used in the Pool Elasticity experiments (Section 5.7.2). Increase starts with a low-load workload ($\mu_i = 0.1, \lambda_i = 10, c_i^* = 126$), and then switches sharply to a high-load workload ($\mu_i = 0.1, \lambda_i = 50, c_i^* = 545$). The additional app, Static, maintains a medium workload the entire time ($\mu_i = 0.1, \lambda_i = 30, c_i^* = 338$). This creates enough diversity in workloads that the pool balance technique is able to contribute. I additionally run the system modeling worker abandonment ($\alpha = 0.1$), recruitment latency ($r = 0.1$), and training latency ($tr_i = 1/60 = 0.017$).

Figures 5.7a-d illustrate how the two systems react to the workloads. In every case, Cioppino offers a significant improvement over the baseline system. Because rule-based pool stability is reactive and inflexible, the baseline has trouble keeping the pool size constant in response to worker abandonment, especially across the varying workloads. Overall, Cioppino shows a 36% higher throughput, 28× lower cost, and 2.5× higher net worker preference than the baseline system.

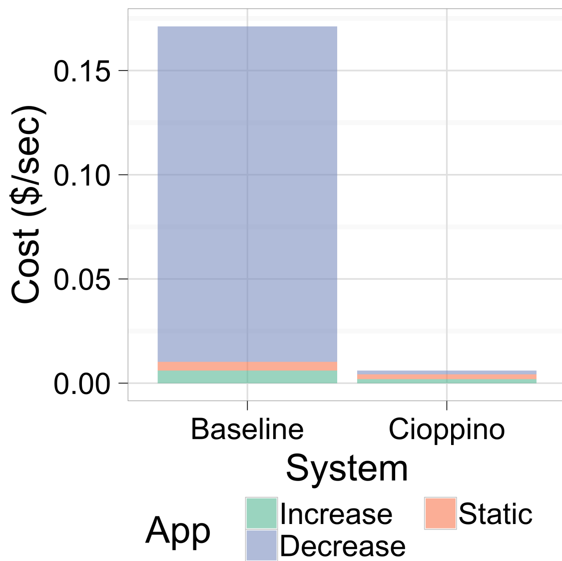
Trace-based data. In addition to the generated behavioral data, I evaluate the two end-to-end systems on the CLAMShell trace described in Section 5.7.1. To keep the evaluation consistent with the fully synthetic experiments, I use the same app workloads: Increase, Decrease, and Static. However, since the CLAMShell trace contains task completion time data, I use CLAMShell completion times instead of generating them from an exponential



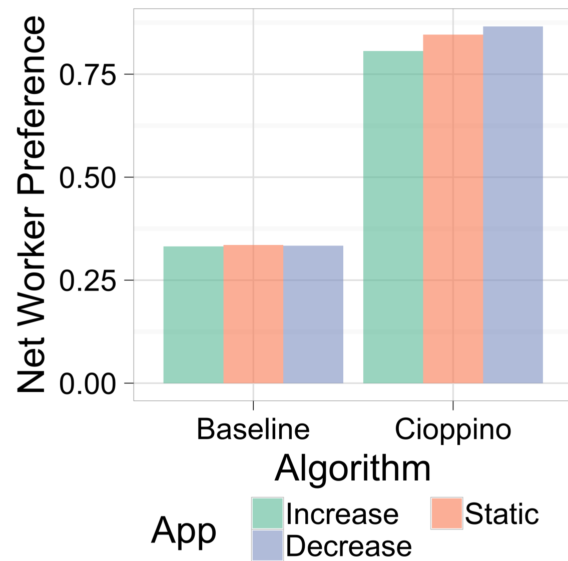
(a) Pool size



(b) Average throughput

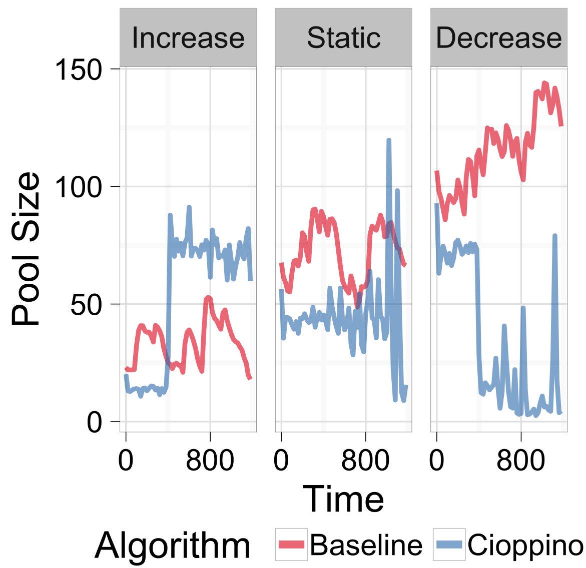


(c) Average cost

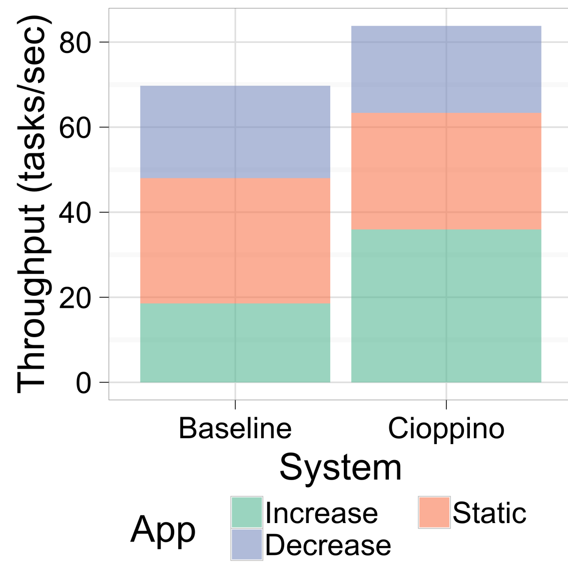


(d) Net worker preference

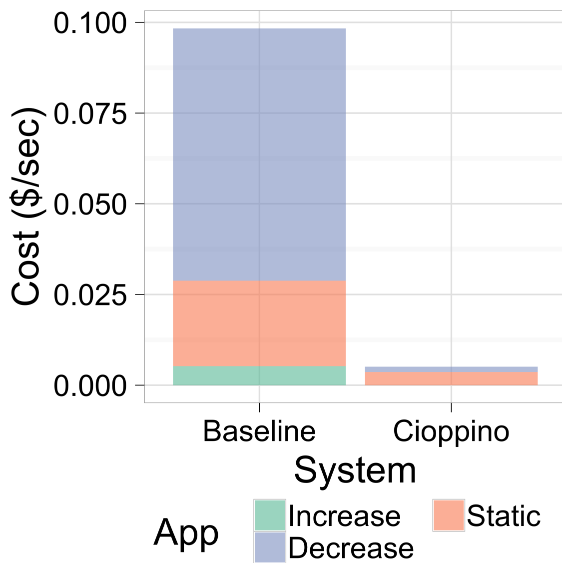
Figure 5.7: Simulated end to end system comparison, averaged over 50 runs.



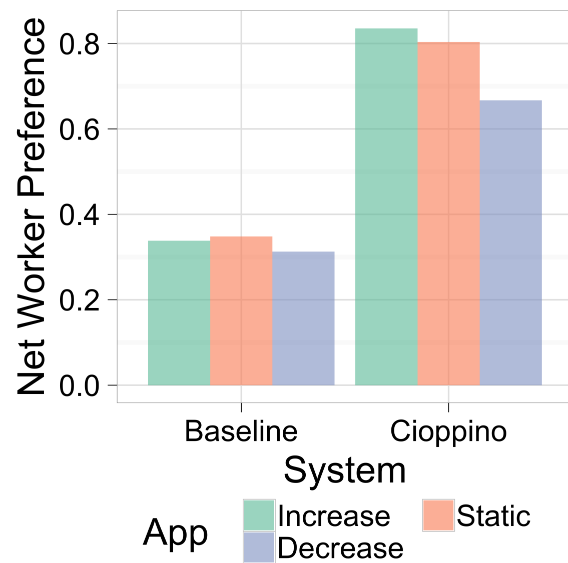
(a) Pool size



(b) Average throughput



(c) Average cost



(d) Net worker preference

Figure 5.8: End to end system comparison, executed against the CLAMShell trace.

distribution given μ_i. In the CLAMShell trace, the average task duration is 1.97 seconds, making $\mu_i = 0.51$. Combined with the λ_i for the app workloads, I compute that c_i^* is 28 for the Increase app, 119 for the Decrease app, and 76 for the static app, so I use these values as starting pool sizes. Additionally, the CLAMShell trace is used to provide worker abandonment times instead of the workload parameter α, and recruitment times instead of the workload parameter r.

Figures 5.8a-d show the techniques' performance on the trace. In the results, we observe similar trends as with the synthetic end to end evaluation. Qualitatively, the Cioppino system is much better at keeping the pool size close to the optimal value in spite of changing workloads (though there is increased noise now that the data is real, not generated). Quantitatively, Cioppino dominates Baseline on all metrics in the cases examined: it has **20% higher throughput**, **19.4× lower cost**, and **2.3× higher net worker preference**.

Summary: *Cioppino improves significantly over the state of the art in managing crowd pools, both on auto-generated and real trace-based crowd behavioral data. It simultaneously improves throughput and net worker preference while drastically reducing cost.*

5.8 Conclusion

In this chapter, I introduced Cioppino, a system for holistic crowd pool management that reduces cost and increases efficiency while accounting for human factors that affect system performance. Cioppino builds on the CLAMShell system of Chapter 4, models crowd task processing as a queueing system, and introduces novel algorithms for autoscaling, compensating for worker abandonment, and rebalancing workers across application pools. Multi-tenant crowd systems should incorporate these techniques to make the most of their crowdsourced workforce.

Together, CLAMShell and Cioppino represent an architecture for designing performant human-powered systems for data analysis that support multiple applications running in parallel on elastic worker pools. However, to maximize performance, both systems need access to a pool of fast workers. Cioppino is concerned with choosing when to hire, release, or transfer workers between pools, but it does not specify *which* individual workers to transfer or hire and therefore provides no insights into identifying fast workers. The Pool Maintenance technique presented in Chapter 4 converges on a pool of fast workers, but relies on a user-defined threshold to separate 'fast' workers from 'slow' ones, and provides no guarantees on how quickly fast workers can be identified. In the next chapter, I investigate the identification of fast workers in an anonymous population with no *a priori* information about their speeds, and prove strong guarantees on the time required for their identification.

Chapter 6

Identifying Fast Workers in Unknown Populations

In Chapters 4 and 5, I described systems that accelerate human-powered data analysis to near-machine timescales, and demonstrated via simulated and real crowd experiments on MTurk that the techniques are effective. Though MTurk workers are a sensible choice for evaluation due to the platform's popularity and accessibility, empirical experiments cannot make high-probability guarantees as to how the systems would perform on other worker populations, an important consideration as crowd platforms evolve and companies increasingly rely on crowds of in-house employees.

Instead, in this chapter I rely on statistical theory to describe the behavior of crowd systems with access to arbitrary worker populations. I describe a key building block of performant crowd systems, the ability to recruit fast workers, in terms of the theoretical framework of a statistical problem known as the most biased coin problem. I then analyze the framework to characterize the difficulty of identifying a fast worker in an unknown population, and design provably near-optimal algorithms for doing so. Along the way, I establish connections between the most biased coin problem and statistical anomaly detection that have important consequences for the analysis of sequential decision processes in general.

6.1 Introduction

Human-powered systems cannot run without humans to power them, and the choice of which workers to recruit can heavily impact the performance of the system. The ultimate goal is for the system to converge on and maintain a pool of fast workers who will consistently process incoming data quickly. An efficient system will arrive at such a pool as fast and cost-effectively as possible.

Because the system can only recruit and send tasks to so many workers at once, identifying fast workers requires navigating an interesting tradeoff between the breadth and depth of the system's exploration. On the one hand, the system needs to improve its confidence in

the speeds of workers that have already been recruited, so that it doesn't erroneously keep slow workers around. On the other hand, the system would like to explore the population for new, faster workers to improve the overall speed of the pool and learn more about the speed distribution of the worker population. Unfortunately, the speeds of workers can only be observed by assigning them tasks, so the system must carefully allocate its resources between deeper exploration of known workers and broader exploration of the unknown population.

In Chapter 4, I described a technique called pool maintenance that converges on a pool of fast workers by repeatedly releasing slow ones and recruiting unknown ones. Pool maintenance navigates the breadth-depth tradeoff using a user-defined cutoff threshold for workers' average latency. Workers are repeatedly assigned tasks until their average latency is observed to be above the threshold with high confidence (depth), then they are released from the pool to be replaced with new workers (breadth). Though I showed that pool maintenance can be effective in practice, setting the threshold requires significant knowledge of both the worker population and the task being performed, and provides no guarantees on how quickly the system can converge on a pool of fast workers.

This chapter explicitly models the breadth-depth tradeoff of fast worker identification using a statistical setup called the most biased coin problem [28]. Focusing on the setting where (as on a crowd platform with an unknown population of workers) the distribution of worker speeds is unknown, I prove novel lower bounds on the number of observed tasks necessary to identify fast workers. Then, I describe algorithms that are guaranteed to identify fast workers with high probability using at most a log factor more tasks than the lower bounds require. Because the analysis is kept general, these algorithms are useful not only for identifying fast workers, but for other important problems unrelated to the topic of this dissertation such as anomaly and intrusion detection and discovery of vacant frequencies in the radio spectrum.

6.1.1 The Most Biased Coin Problem

The most biased coin problem uses a simple setup to describe the breadth-depth tradeoff of exploration. Consider a bag that contains an infinite number of two kinds of biased coins: "heavy" coins with mean $\theta_1 \in (0, 1)$ and "light" coins with mean $\theta_0 \in (0, \theta_1)$. When a player picks a coin from the bag, with probability α the coin is "heavy" and with probability $(1-\alpha)$ the coin is "light." The player can flip any coin she picks from the bag as many times as she wants, and the goal is to identify a heavy coin using as few total flips as possible. When $\alpha, \theta_0, \theta_1$ are unknown, the key difficulty of this problem lies in distinguishing whether the two kinds of coins have very similar means, or whether heavy coins are just extremely rare. That is, one must balance flipping an individual coin many times to better estimate its mean against considering many new coins to maximize the probability of observing a heavy one.

In the same way, crowd systems must balance sending tasks to individual workers many times to better estimate their speed against hiring many new workers to maximize the probability of observing a fast one. As a result, solutions to the most biased coin problem can be used to identify fast workers on crowdsourcing platforms. Doing so makes an important

simplifying assumption about what is meant by a 'fast' worker, however. Instead of having a cutoff latency that separates fast workers from slow workers, the most biased coin approach assumes that workers are either fast (with mean speed θ_1) or slow (with mean speed θ_0). This assumption, though restrictive, allows for more powerful analysis of the algorithms, which can still be used to identify workers whose mean speed deviates significantly from the population mean. The most biased coin problem also assumes that coin means do not change over time, which may not be true of crowd workers who get faster as they become more familiar with the work. However, the algorithms in this chapter could be extended to handle this in a straightforward fashion by periodically 'forgetting' about a worker's previously completed tasks and learning about them anew.

Previous work has only proposed solutions to the most biased coin problem that rely on some or full knowledge of $\alpha, \theta_0, \theta_1$, limiting their applicability. This work proposes the first algorithm that requires no knowledge of $\alpha, \theta_0, \theta_1$, is guaranteed to return a heavy coin with probability at least $1-\delta$, and flips a total number of coins, in expectation, that nearly matches known lower bounds. Moreover, the fully adaptive algorithm proposed here supports sub-Gaussian sources in addition to just coins (important because worker speeds are not observed as heads or tails, but as real numbers), and only ever has one "coin" outside the bag at a given time (important because a crowd system cannot send tasks to workers not currently in the pool).

In order to derive bounds on the most biased coin problem, I connect it to anomaly detection, so these results are useful not only to crowdsourcing systems, but to any application that seeks to detect the presence of a mixture versus just a single component of a known family of distributions (e.g. $X \sim (1-\alpha)g_{\theta_0} + \alpha g_{\theta_1}$ versus $X \sim g_\theta$ for some θ). I show that in detecting the presence of a mixture distribution, there is a stark difference in difficulty between when the underlying distribution parameters are known (e.g. $\alpha, \theta_0, \theta_1$) and when they are not. The most biased coin problem can be viewed as an online, adaptive mixture detection problem where source distributions arrive one at a time that are either g_{θ_0} with probability $(1-\alpha)$ or g_{θ_1} with probability α (e.g. null or anomolous) and the player adaptively chooses how many samples to take from each distribution (to increase the signal-to-noise ratio) with the goal of identifying an anomolous distribution f_{θ_1} using as few total number of samples as possible. One consequence of this work is drawing a contrast between the power of adaptive versus non-adaptive (e.g. taking the same number of samples each time) approaches to this problem, specifically when $\alpha, \theta_0, \theta_1$ are unknown.

6.1.2 Related Work

The most biased coin problem was first proposed by Chandrasekaran et al. [28]. In that work, it was shown that if $\alpha, \theta_0, \theta_1$ were known then there exists an algorithm based on the sequential probability ratio test (SPRT) that is optimal in that it minimizes the expected number of total flips to find a "heavy" coin whose posterior probability of being heavy is at

least $1-\delta$, and the expected sample complexity of this algorithm was upper-bounded by

$$\frac{16}{(\theta_1-\theta_0)^2}\left(\frac{1-\alpha}{\alpha}+\log\left(\frac{(1-\alpha)(1-\delta)}{\alpha\delta}\right)\right). \tag{6.1}$$

However, the practicality of the proposed algorithm is severely limited as it relies critically on knowing α, θ_0, and θ_1 exactly. In addition, the algorithm returns to coins it has previously flipped and thus requires more than one coin to be outside the bag at a time, ruling out some applications. Malloy et al. [98] addressed some of the shortcomings of [29] (a preprint of [28]) by considering both an alternative SPRT procedure and a sequential thresholding procedure. Both of these proposed algorithms only ever have one coin out of the bag at a time. However, the former requires knowledge of all relevant parameters $\alpha, \theta_0, \theta_1$, and the latter requires knowledge of α, θ_0. Moreover, these results are only presented for the asymptotic case where $\delta \to 0$.

The most biased coin problem can be viewed through the lens of multi-armed bandits. In the best-arm identification problem, the player has access to K distributions (arms) such that if arm $i \in [K]$ is sampled (pulled), an iid random variable with mean μ_i is observed; the objective is to identify the arm associated with the highest mean with probability at least $1-\delta$ using as few pulls as possible (see [77] for a short survey). In the *infinite* armed bandit problem, the player is not confined to K arms but an infinite reservoir of arms such that a draw from this reservoir results in an arm with a mean μ drawn from some distribution; the objective is to identify the highest mean possible after n total pulls for any $n > 0$ with probability $1-\delta$ (see [25]). The most biased coin problem is an instance of this latter game with the arm reservoir distribution of means μ defined as $\mathbb{P}(\mu \geq \theta_1 - \epsilon) = \alpha \mathbf{1}_{\epsilon \geq 0} + (1-\alpha)\mathbf{1}_{\epsilon \geq \theta_1 - \theta_0}$ for all ϵ. Previous work has focused on an alternative arm distribution reservoir that satisfies $E\epsilon^\beta \leq \mathbb{P}(\mu \geq \mu_* - \epsilon) \leq E'\epsilon^\beta$ for some $\mu_* \in [0,1]$ where E, E' are constants and β is known [19, 145, 21, 25]. Because neither arm distribution reservoir can be written in terms of the other, neither work subsumes the other. Note that one can always apply an algorithm designed for the infinite armed bandit problem to any finite K-armed bandit problem by defining the arm reservoir as placing a uniform distribution over the K arms. This is appealing when K is very large and one wishes to guarantee nontrivial performance when the number of pulls is much less than K[1]. The most biased coin problem is a special case of the K-armed reservoir distribution where one arm has mean θ_1 and $K-1$ arms have mean θ_0 with $\alpha = \frac{1}{K}$.

Given that [28] and [98] are provably optimal algorithms for the most biased coin problem given knowledge of $\alpha, \theta_0, \theta_1$, it is natural to consider a procedure that first estimates these unknown parameters first and then uses these estimates in the algorithms of [28] or [98]. Indeed, in the β-parameterized arm reservoir setting discussed above, this is exactly what [25] propose to do, suggesting a particular estimator for β given a lower bound $\widehat{\beta} \leq \beta$. They show that this estimator is sufficient to obtain the same sample complexity result up to log

[1] All current algorithms for the K-armed bandit problem begin by sampling each arm once so that until the number of pulls exceeds K, performance is no better than random selection.

factors as when β was known. Sadly, through upper and lower bounds I show that for the most biased coin problem this *estimate-then-explore* approach requires quadratically more flips than our proposed algorithm that adapts to these unknown parameters. Specifically, when $\theta_1 - \theta_0$ is sufficiently small, one cannot use a static estimation step to determine whether $\alpha = 0$ or $\alpha > 0$ unless a number of samples *quadratic* in the optimal sample complexity are taken.

This chapter's contributions to the most biased coin problem include a novel algorithm that never has more than one coin outside the bag at a time, has no knowledge of the distribution parameters, supports distributions on $[0, 1]$ rather than just "coins," and comes within log factors of the known information-theoretic lower bound and Equation 6.1 which is achieved by an algorithm that knows the parameters. See Table 6.1 for an overview of the upper and lower bounds proved in this work for this problem. This algorithm is the first solution to the most biased coin problem that does not require prior knowledge of the problem parameters, and its approach can be reworked to solve more general instances of the infinite-armed bandit problem, including the β-parameterized and K-armed reservoir cases described above. Finally, if an algorithm is desired for arbitrary arm reservoir distributions, this work rules out an estimate-then-explore approach.

6.1.3 Problem Statement

Let $\theta \in \Theta$ index a family of single-parameter probability density functions g_θ and fix $\theta_0, \theta_1 \in \Theta$, $\alpha \in [0, 1/2]$. For any $\theta \in \Theta$ assume that g_θ is known to the procedure. Note that in the most biased coin problem, g_θ =Bernoulli(θ), but in general it is arbitrary (e.g. $\mathcal{N}(\theta, 1)$). Consider a sequence of iid Bernoulli random variables $\xi_i \in \{0, 1\}$ for $i = 1, 2, \ldots$ where each $\mathbb{P}(\xi_i = 1) = 1 - \mathbb{P}(\xi_i = 0) = \alpha$. Let $X_{i,j}$ for $j = 1, 2, \ldots$ be a sequence of random variables drawn from g_{θ_1} if $\xi_i = 1$ and g_{θ_0} otherwise, and let $\{\{X_{i,j}\}_{j=1}^{M_i}\}_{i=1}^{N}$ represent the sampling history generated by a procedure for some $N \in \mathbb{N}$ and $(M_1, \ldots, M_N) \in \mathbb{N}^N$. Any valid procedure behaves according to Algorithm 6.1:

Algorithm 6.1 The most biased coin procedure. Only the last distribution drawn may be sampled or declared heavy, enforcing the rule that only one coin may be outside the bag at a time.

Initialize an empty history $(N = 1, M = (0, 0, \ldots))$.
Repeat until heavy distribution declared:
Choose one of

1. draw a sample from distribution N, $M_N \leftarrow M_N + 1$
2. draw a sample from the $(N + 1)$st distribution, $M_{N+1} = 1$, $N \leftarrow N + 1$
3. declare distribution N as heavy

Definition 1 *A strategy for the most biased coin problem is* δ**-probably correct** *if for all* $(\alpha, \theta_0, \theta_1)$ *it identifies a "heavy"* g_{θ_1} *distribution with probability at least* $1 - \delta$*.*

Definition 2 (Strategies for the most biased coin problem) *An* **estimate-then-explore strategy** *is a strategy that, for any fixed $m \in \mathbb{N}$, begins by sampling each successive coin exactly m times for a number of coins that is at least the minimum necessary for any test to determine that $\alpha \neq 0$ with probability at least $1-\delta$, then optionally continues sampling with an arbitrary strategy that declares a heavy coin. An* **adaptive strategy** *is any strategy that is not an* estimate-then-explore *strategy.*

Estimate-then-explore strategies are worth considering because there exist optimal algorithms [28, 98] for the most biased coin problem if $\alpha, \theta_0, \theta_1$ are known, so it is natural to consider estimating these quantities then using one of these algorithms. Note that the algorithm of [25] for the β-parameterized infinite armed bandit problem discussed above can be considered an *estimate-then-explore strategy* since it first estimates β by sampling a fixed number of samples from a set of arms, and then uses this estimate to draw a fixed number of arms and applies a UCB-style algorithm to these arms. A contribution of this work is showing that such a strategy is infeasible for the most biased coin problem.

For all strategies that are δ-probably correct and follow the interface of Algorithm 6.1, the goal is to provide lower and upper bounds on the quantity $\mathbb{E}[T] := \mathbb{E}[\sum_{i=1}^{N} M_i]$ for any $(\alpha, \theta_0, \theta_1)$ if N denotes the final number of coins considered.

6.2 From Identifying Coins to Detecting Mixture Distributions

Addressing the most biased coin problem, Malloy et al. analyze perhaps the most natural strategy: fix an $m \in \mathbb{N}$ and flip each successive coin exactly m times [98]. The relevant questions are how large does m have to be in order to guarantee correctness with probability $1-\delta$, and for a given m how long must one wait to declare a "heavy" coin? The authors partially answer these questions and Section 6.3.2.1 improves upon them, which leads to further study of the difficulty of detecting the presence of a mixture distribution. As an example of the kind of lower bounds shown in this work, given observations of a sequence of random variables $X_1, \ldots, X_n$, consider the following hypothesis test:

$$\begin{aligned} &\mathbf{H}_0 : \forall i \quad X_1, \ldots, X_n \sim \mathcal{N}(\theta, \sigma^2) \quad \text{for some } \theta \in \mathbb{R}, \\ &\mathbf{H}_1 : \forall i \quad X_1, \ldots, X_n \sim (1-\alpha)\mathcal{N}(\theta_0, \sigma^2) + \alpha\, \mathcal{N}(\theta_1, \sigma^2) \end{aligned} \tag{P1}$$

which will henceforth be referred to as Problem P1 or just (P1). We can show that if $\theta_0, \theta_1, \alpha$ are *known* and $\theta = \theta_0$, then it is *sufficient* to observe just $\max\{1/\alpha, \frac{\sigma^2}{\alpha^2(\theta_1-\theta_0)^2}\log(1/\delta)\}$ samples to determine the correct hypothesis with probability at least $1-\delta$. However, if $\theta_0, \theta_1, \alpha$ are *unknown* then it is *necessary* to observe at least $\max\left\{1/\alpha, \left(\frac{\sigma^2}{\alpha(\theta_1-\theta_0)^2}\right)^2\log(1/\delta)\right\}$ samples in expectation whenever $\frac{(\theta_1-\theta_0)^2}{\sigma^2} \leq 1$ and $\max\{1/\alpha, \frac{\sigma^2}{\alpha^2(\theta_1-\theta_0)^2}\log(1/\delta)\}$ otherwise (see Section 6.4).

Recognizing $\frac{(\theta_1-\theta_0)^2}{\sigma^2}$ as the KL divergence between two Gaussians of $\mathbf{H}_1$, there are startling consequences for anomaly detection when the parameters of the underlying distributions are unknown: if the anomalous distribution is well separated from the null distribution, then detecting an anomalous component is only about as hard as observing just one anomalous sample (i.e. $1/\alpha$) multiplied by the inverse KL divergence between the null and anomalous distributions. However, when the two distributions are *not* well separated then the necessary sample complexity explodes to this latter quantity *squared*. In Section 6.5 we will investigate adaptive methods for dramatically decreasing this sample complexity.

The lower bounds in this chapter are based on the detection of the presence of a mixture of two distributions of an exponential family versus just a single distribution of the same family. There has been extensive work in the estimation of mixture distributions [69, 53] but this literature often assumes that the mixture coefficient α is bounded away from 0 and 1 to ensure a sufficient number of samples from each distribution. In contrast, this work highlights the regime when α is arbitrarily small, as is the case in statistical anomaly detection [51, 135, 3]. Property testing, e.g. unimodality, [1] is relevant but can lack interpetability or strength in favor of generality. Considering the exponential family allows for interpretable statements about the relevant problem parameters in different regimes.

Preliminaries Let P and Q be two probability distributions with densities p and q, respectively. For simplicity, assume p and q have the same support. Define the *KL Divergence* between P and Q as $KL(P,Q) = \int \log\left(\frac{p(x)}{q(x)}\right) dp(x)$. Define the χ^2 *Divergence* between P and Q as $\chi^2(P,Q) = \int \left(\frac{p(x)}{q(x)} - 1\right)^2 dq(x) = \int \frac{(p(x)-q(x))^2}{q(x)} dx$. Note that by Jensen's inequality

$$KL(P,Q) = \mathbb{E}_p\left[\log\left(\tfrac{p}{q}\right)\right] \leq \log\left(\mathbb{E}_p\left[\tfrac{p}{q}\right]\right) = \log\left(\chi^2(P,Q)+1\right) \leq \chi^2(P,Q). \tag{6.2}$$

Examples: If $P = \mathcal{N}(\theta_1,\sigma^2)$ and $Q = \mathcal{N}(\theta_0,\sigma^2)$ then $KL(P,Q) = \frac{(\theta_1-\theta_0)^2}{2\sigma^2}$ and $\chi^2(P,Q) = e^{\frac{(\theta_1-\theta_0)^2}{\sigma^2}} - 1$. If $P = \text{Bernoulli}(\theta_1)$ and $Q = \text{Bernoulli}(\theta_0)$ then $KL(P,Q) = \theta_1 \log(\frac{\theta_1}{\theta_0}) + (1-\theta_1)\log(\frac{1-\theta_1}{1-\theta_0}) \leq \frac{(\theta_1-\theta_0)^2/2}{\theta_0(1-\theta_0)-[(\theta_1-\theta_0)(2\theta_0-1)]_+}$ and $\chi^2(P,Q) = \frac{(\theta_1-\theta_0)^2}{\theta_0(1-\theta_0)}$.

Proofs of theorems are omitted from this chapter for clarity of exposition, and can be found in Appendix A.

6.3 Lower bounds

This section presents lower bounds on the sample complexity of δ-*probably correct* strategies for the most biased coin problem that follow the interface of Algorithm 6.1. Lower bounds are stated for any *adaptive* strategy in Section 6.3.1, non-adaptive strategies that may have knowledge of the parameters but sample each distribution the same number of times in Section 6.3.2.1, and *estimate-then-explore* strategies that do not have prior knowledge of the parameters in Section 6.3.2.2. These lower bounds, with the exception of the adaptive

strategy, are based on the difficulty of detecting the presence of a mixture distribution, and this reduction is explained in Section 6.3.2.

6.3.1 Adaptive strategies

The following theorem, reproduced from [98], describes the sample complexity of any δ-probably correct algorithm for the most biased coin identification problem. Note that this lower bound holds for any procedure even if it returns to previously seen distributions to draw additional samples and even if it knows $\alpha, \theta_0, \theta_1$.

Theorem 1 *[98, Theorem 2] Fix $\delta \in (0,1)$. Let T be the total number of samples taken of any procedure that is δ-probably correct in identifying a heavy distribution. Then*

$$\mathbb{E}[T] \geq c_1 \max\left\{\frac{1-\delta}{\alpha}, \frac{(1-\delta)}{\alpha KL(g_{\theta_0}|g_{\theta_1})}\right\}$$

whenever $\alpha \leq c_2\delta$ where $c_1, c_2 \in (0,1)$ are absolute constants.

The above theorem is directly applicable to the special case where g_θ is a Bernoulli distribution, implying a lower bound of $\max\left\{\frac{1-\delta}{\alpha}, \frac{2\min\{\theta_0(1-\theta_0),\theta_1(1-\theta_1)\}}{\alpha(\theta_1-\theta_0)^2}\right\}$ for the most biased coin problem. The upper bounds of this work's proposed procedures for the most biased coin problem presented later will be compared to this benchmark.

6.3.2 The detection of a mixture distribution and the most biased coin problem

First observe that identifying a specific distribution $i \leq N$ as heavy (i.e. $\xi_i = 1$) or determining that α is strictly greater than 0, is at least as hard as detecting that *any* of the distributions up to distribution N is heavy. Thus, a lower bound on the total expected number of samples of all considered distributions for this strictly easier detection problem is also a lower bound for the estimate-then-explore strategy for the most biased coin identification problem.

The estimate-then-explore strategy fixes an $m \in \mathbb{N}$ prior to starting the game and then samples each distribution exactly m times, i.e. $M_i = m$ for all $i \leq N$ for some N. To simplify notation let f_θ denote the distribution of the sufficient statistics of these m samples. In general f_θ is a product distribution, but when g_θ is a Bernoulli distribution, as in the biased coin problem, we can take f_θ to be a Binomial distribution with parameters (m, θ). Now the problem is more succinctly described as:

$$\begin{aligned}
&\mathbf{H}_0 : \forall i \quad X_i \sim f_\theta \qquad \text{for some } \theta \in \widetilde{\Theta} \subseteq \Theta, \\
&\mathbf{H}_1 : \forall i \quad \xi_i \sim \text{Bernoulli}(\alpha), \quad \forall i \quad X_i \sim \begin{cases} f_{\theta_0} & \text{if } \xi_i = 0 \\ f_{\theta_1} & \text{if } \xi_i = 1 \end{cases}
\end{aligned} \tag{P2}$$

If θ_0 and θ_1 are close to each other, or if α is very small, it can be very difficult to decide between $\mathbf{H}_0$ and $\mathbf{H}_1$ even if $\alpha, \theta_0, \theta_1$ are known a priori. Note that when the parameters are *known*, one can take $\widetilde{\Theta} = \{\theta_0\}$. However, when the parameters are *unknown*, one takes $\widetilde{\Theta} = \Theta$ to prove a lower bound on the sample complexity of the estimate-then-explore algorithm, which is tasked with deciding whether or not samples are coming from a mixture of distributions or just a single distribution within the family. That is, lower bounds on the sample complexity when the parameters are known and unknown follow by analyzing a simple binary and composite hypothesis test, respectively. In what follows, for any event A, let $\mathbb{P}_i(A)$ and $\mathbb{E}_i[A]$ denote probability and expectation of A under hypothesis $\mathbf{H}_i$ for $i \in \{0, 1\}$ (the specific value of θ in $\mathbf{H}_0$ will be clear from context). The next claim is instrumental in proving lower bounds on the difficulty of the hypothesis tests.

Claim 1 *Any procedure that is δ-probably correct also satisfies $\mathbb{P}_0(N < \infty) \leq \delta$ whenever $\alpha = 0$.*

Proof Suppose there exists a δ-probably correct procedure with $\mathbb{P}(N < \infty) > \delta$ on a problem instance $(\alpha, \theta_0, \theta_1)$ when $\alpha = 0$. Then there exists a finite $\hat{n} \in \mathbb{N}$ such that $\mathbb{P}(N \leq \hat{n}) > \delta$. For some $\epsilon \in (0, 1)$ to be defined later, define $\hat{\alpha} = \frac{\log(\frac{1}{1-\epsilon})}{2\hat{n}}$ and note that for this $\hat{\alpha}$, $\mathbb{P}(\bigcap_{i=1}^{\hat{n}} \{\xi_i = 0\}) = (1 - \hat{\alpha})^{\hat{n}} \geq e^{-2\hat{n}\hat{\alpha}} \geq 1 - \epsilon$. Thus, the probability that the procedure terminates with a light distribution under $\alpha = \hat{\alpha}$ is at least

$$\begin{aligned}
\mathbb{P}_{\alpha=\hat{\alpha}}(N \leq \hat{n}, \cap_{i=1}^{\hat{n}} \{\xi_i = 0\}) &= \mathbb{P}_{\alpha=\hat{\alpha}}(N \leq \hat{n} | \cap_{i=1}^{\hat{n}} \{\xi_i = 0\}) \mathbb{P}_{\alpha=\hat{\alpha}}(\cap_{i=1}^{\hat{n}} \{\xi_i = 0\}) \\
&= \mathbb{P}_{\alpha=0}(N \leq \hat{n}) \mathbb{P}_{\alpha=\hat{\alpha}}(\cap_{i=1}^{\hat{n}} \{\xi_i = 0\}) > \delta(1 - \epsilon).
\end{aligned}$$

Because we can make ϵ arbitrarily small, the above display implies that the procedure makes a mistake with probability at least δ, but this is a contradiction as the procedure is δ-probably correct. ■

6.3.2.1 Sample complexity when parameters are known

Theorem 2 *Fix $\delta \in (0, 1)$. Consider the hypothesis test of Problem P2 for any fixed $\theta \in \widetilde{\Theta} \subseteq \Theta$. Let N be the random number of distributions considered before stopping and declaring a hypothesis. If a procedure satisfies $\mathbb{P}_0(N < \infty) \leq \delta$ and $\mathbb{P}_1(\cup_{i=1}^{N} \{\xi_i = 1\}) \geq 1 - \delta$, then $\mathbb{E}_1[N] \geq \max\left\{\frac{1-\delta}{\alpha}, \frac{\log(1/\delta)}{KL(\mathbb{P}_1 | \mathbb{P}_0)}\right\} \geq \max\left\{\frac{1-\delta}{\alpha}, \frac{\log(1/\delta)}{\chi^2(\mathbb{P}_1 | \mathbb{P}_0)}\right\}$. In particular, if $\widetilde{\Theta} = \{\theta_0\}$ then*

$$\mathbb{E}_1[N] \geq \max\left\{\frac{1-\delta}{\alpha}, \frac{\log(1/\delta)}{\alpha^2 \chi^2(f_{\theta_1} | f_{\theta_0})}\right\}.$$

The next corollary relates Theorem 2 to the most biased coin problem and is related to [98, Theorem 4] that considers the limit as $\alpha \to 0$ and assumes m is sufficiently large (specifically, large enough for the Chernoff-Stein lemma to apply). In contrast, this result holds for all finite δ, α, m.

Corollary 1 *Fix $\delta \in (0,1)$. For any $m \in \mathbb{N}$ consider a δ-probably correct strategy that flips each coin exactly m times. If N_m is the number of coins considered before declaring a coin as heavy then*

$$\min_{m \in \mathbb{N}} \mathbb{E}[mN_m] \geq \frac{(1-\delta)\log\left(\frac{\log(1/\delta)}{\alpha}\right)}{\alpha} \frac{\theta_0(1-\theta_0)}{(\theta_1-\theta_0)^2}.$$

One can show the existence of such a strategy with a nearly matching upperbound when $\alpha, \theta_0, \theta_1$ are known (see Section 6.6.1). Note that this is at least $\log(1/\alpha)$ larger than the sample complexity of (6.1) that can be achieved by an adaptive algorithm when the parameters are known.

6.3.2.2 Sample complexity when parameters are unknown

If α, θ_0, and θ_1 are unknown, f_{θ_0} cannot be tested against the mixture $(1-\alpha)f_{\theta_0} + \alpha f_{\theta_1}$. Instead, the challenge is to perform the general composite test of *any* individual distribution against *any* mixture, which is at least as hard as the hypothesis test of Problem P2 with $\widetilde{\Theta} = \{\theta\}$ for some particular worst-case setting of θ. Without any specific form of f_θ, it is difficult to pick a worst case θ that will produce a tight bound. Consequently, in this section I consider single parameter exponential families (defined formally below) to provide a class of distributions in which it is easy to reason about different possible values for θ. Since exponential families include Bernoulli, Gaussian, exponential, and many other distributions, the following theorem is general enough to be useful in a wide variety of settings. The constant C referred to in the next theorem is an absolute constant under certain conditions outlined in the following remark and corollary. Its explicit form is given in the proof.

Theorem 3 *Suppose f_θ for $\theta \in \Theta \subset \mathbb{R}$ is a single parameter exponential family so that $f_\theta(x) = h(x)\exp(\eta(\theta)x - b(\eta(\theta)))$ for some scalar functions h, b, η where η is strictly increasing. If $\widetilde{\Theta} = \{\theta_*\}$ where $\theta_* = \eta^{-1}\big((1-\alpha)\eta(\theta_0) + \alpha\eta(\theta_1)\big)$ and N is the stopping time of any procedure that satisfies $\mathbb{P}_0(N < \infty) \leq \delta$ and $\mathbb{P}_1(\cup_{i=1}^N \{\xi_i = 1\}) \geq 1-\delta$, then*

$$\mathbb{E}_1[N] \geq \max\left\{\tfrac{1-\delta}{\alpha}, \tfrac{\log(\frac{1}{\delta})}{C\left(\frac{1}{2}\alpha(1-\alpha)(\eta(\theta_1)-\eta(\theta_0))^2\right)^2}\right\}.$$

where C is a constant that may depend on $\alpha, \theta_0, \theta_1$.

The following remark and corollary apply Theorem 3 to the special cases of Gaussian mixture model detection and the most biased coin problem, respectively.

Remark 1 *When $\alpha, \theta_0, \theta_1$ are unknown, any procedure has no knowledge of $\widetilde{\Theta}$ in Problem P2 and consequently it cannot rule out $\theta = \theta_*$ for $\mathbf{H}_0$ where θ_* is defined in Theorem 3. If $f_\theta = \mathcal{N}(\theta, \sigma^2)$ for known σ, then whenever $\frac{(\theta_1-\theta_0)^2}{\sigma^2} \leq 1$ the constant C in Theorem 3 is an absolute constant and consequently, $\mathbb{E}_1[N] = \Omega\big(\big(\frac{\sigma^2}{\alpha(\theta_1-\theta_0)^2}\big)^2 \log(1/\delta)\big)$. Conversely, when*

$\alpha, \theta_0, \theta_1$ are known, Problem P2 reduces to determining whether samples came from $\mathcal{N}(\theta_0, \sigma^2)$ or $(1-\alpha)\mathcal{N}(\theta_0, \sigma^2) + \alpha\mathcal{N}(\theta_1, \sigma^2)$, and it is sufficient to take just $O\left(\frac{\sigma^2}{\alpha^2(\theta_1-\theta_0)^2}\log(1/\delta)\right)$ samples (see Section 6.4).

Corollary 2 *Fix $\delta \in [0,1]$ and assume θ_0, θ_1 are bounded sufficiently far from $\{0,1\}$ such that $2(\theta_1-\theta_0) \leq \min\{\theta_0(1-\theta_0), \theta_1(1-\theta_1)\}$. For any m let N_m be the number of coins a δ-probably correct estimate-then-explore strategy that flips each coin m times in the exploration step. Then*

$$m\mathbb{E}[N_m] \geq \frac{c' \min\{\frac{1}{m}, \theta_*(1-\theta_*)\}}{\left(\alpha(1-\alpha)\frac{(\theta_1-\theta_0)^2}{\theta_*(1-\theta_*)}\right)^2}\log(\tfrac{1}{\delta}) \quad \text{whenever} \quad m \leq \frac{\theta_*(1-\theta_*)}{(\theta_1-\theta_0)^2}.$$

where c' is an absolute constant and $\theta_* = \eta^{-1}\left((1-\alpha)\eta(\theta_0) + \alpha\eta(\theta_1)\right) \in [\theta_0, \theta_1]$.

Remark 2 *If $\alpha, \theta_0, \theta_1$ are unknown, any estimate-then-explore strategy (or the strategy described in Corollary 1) would be unable to choose an m that depended on these parameters, so it can be treated it as a constant. Thus, for the case when θ_0 and θ_1 are bounded away from $\{0,1\}$ (e.g. $\theta_0, \theta_1 \in [1/8, 7/8]$), the above corollary states that for any fixed m, whenever $\theta_1 - \theta_0$ is sufficiently small the number of samples necessary for these strategies to identify a heavy coin scales like $\left(\frac{1}{\alpha(\theta_1-\theta_0)^2}\right)^2\log(1/\delta)$. This is striking example of the difference when parameters are known versus when they are not and effectively rules out an estimate-then-explore strategy for practical purposes.*

6.4 A Discussion of Gaussians

The theorems thus far have been stated as generally as possible to permit them to apply to applications, for example crowd worker identification, where no assumptions are made regarding the distributions of individual coins (or workers). In this section, I relate the specific setting of Problem P1, where individual distributions are Gaussians, to the general bounds presented so far.

6.4.1 On the detection of a mixture of Gaussians

For known σ^2, consider the hypothesis test of Problem P1. In what follows, let $\chi^2(\theta_1, \theta_0)$ and $KL(\theta_1, \theta_0)$ be the chi-squared and KL divergences of the two distributions of $\mathbf{H}_1$. Note that for $\frac{(\theta_1-\theta_0)^2}{\sigma} \leq 1$, we have that $\chi^2(\theta_1, \theta_0) = e^{\frac{(\theta_1-\theta_0)^2}{\sigma^2}} - 1 \leq 2\frac{(\theta_1-\theta_0)^2}{\sigma^2} = 4KL(\theta_1, \theta_0)$.

Theorem 2 says that for $\frac{(\theta_1-\theta_0)^2}{\sigma^2} \leq 1$, a procedure that has maximum probability of error less than δ requires at least $\max\left\{\frac{1-\delta}{\alpha}, \frac{\log(1/\delta)}{4\alpha^2 KL(\theta_1,\theta_0)}\right\}$ samples to decide the above hypohesis test, even if $\alpha, \theta_0, \theta_1$ are known. The next subsection shows that if $\alpha, \theta_0, \theta_1$ are unknown then one requires at least $\frac{\log(1/\delta)}{2[\alpha KL(\theta_1,\theta_0)]^2}$ samples to decide the above hypothesis test correctly with probability at least $1-\delta$. This is likely achievable using the method of moments [69].

6.4.2 Lower bounds

Theorem 4 *For known σ^2, consider the hypothesis test of Problem P1. If $\theta_* = (1-\alpha)\theta_0 + \alpha\theta_1$ and $\frac{\theta_1-\theta_0}{\sigma} \leq 1$ then*

$$\chi^2((1-\alpha)f_{\theta_0}(x) + \alpha f_{\theta_1}(x) | f_{\theta_*}(x)) \leq c' \left(\alpha(1-\alpha)\frac{(\theta_1-\theta_0)^2}{\sigma^2} \right)^2$$

for some absolute constant c'.

6.4.3 Gaussian Upper bound for known $\alpha, \theta_0, \theta_1$

For known σ^2, consider the hypothesis test of Problem P1 with $\theta = \theta_0$. We observe a sample $X_1, \dots, X_n$ and are trying to establish whether it came from $\mathbf{H}_0$ or $\mathbf{H}_1$.

Consider the test

$$\frac{1}{n}\sum_{i=1}^n \mathbf{1}_{X_i > \theta_1} \underset{\mathbf{H}_0}{\overset{\mathbf{H}_1}{\gtrless}} \frac{\mathbb{P}_1(X_1 > \theta_1) + \mathbb{P}_0(X_1 > \theta_1)}{2} =: \gamma.$$

If $\epsilon = \mathbb{P}_1(X_1 > \theta_1) - \mathbb{P}_0(X_1 > \theta_1)$ then

$$\mathbb{P}_1\left(\frac{1}{n}\sum_{i=1}^n \mathbf{1}_{X_i > \theta_1} \leq \gamma\right) = \mathbb{P}_1\left(\frac{1}{n}\sum_{i=1}^n \mathbf{1}_{X_i > \theta_1} \leq \mathbb{P}_1(X_1 > \theta_1) - \epsilon/2\right) \leq e^{-n\epsilon^2/2}$$

and

$$\mathbb{P}_0\left(\frac{1}{n}\sum_{i=1}^n \mathbf{1}_{X_i > \theta_1} \geq \gamma\right) = \mathbb{P}_0\left(\frac{1}{n}\sum_{i=1}^n \mathbf{1}_{X_i > \theta_1} \geq \mathbb{P}_0(X_1 > \theta_1) + \epsilon/2\right) \leq e^{-n\epsilon^2/2}$$

by sub-Gaussian tail bounds. If $Q(x) = \int_x^\infty \frac{1}{\sqrt{2\pi}} e^{-z^2/2} dz$ and $\Delta = \frac{\theta_1 - \theta_0}{\sigma}$ then

$$\mathbb{P}_0(X_1 > \theta_1) = Q(\Delta)$$
$$\mathbb{P}_1(X_1 > \theta_1) = (1-\alpha)Q(\Delta) + \alpha\frac{1}{2}$$

so

$$\epsilon = \alpha\left(\frac{1}{2} - Q(\Delta)\right) = \alpha\int_0^\Delta \frac{1}{\sqrt{2\pi}} e^{-x^2/2} dx \geq \min\{\frac{\alpha\Delta}{4\sqrt{2\pi}}, \frac{1}{4}\alpha\}.$$

Thus, the test fails with probability at most

$$\exp\left[-n\alpha^2 \min\left\{\frac{(\theta_1-\theta_0)^2}{64\pi\sigma^2}, \frac{1}{32}\right\}\right].$$

We conclude that if $\Delta = \frac{\theta_1-\theta_0}{\sigma} \leq 1$ and $n \geq \frac{(\theta_1-\theta_0)^2 \log(1/\delta)}{64\pi\alpha^2\sigma^2} = \frac{KL(\mathbb{P}_{\theta_1}, \mathbb{P}_{\theta_0})\log(1/\delta)}{64\pi\alpha^2}$ the correct hypothesis is selected. The $1/\alpha$ sufficiency result holds for large enough Δ since one merely needs to observe just one sample since the probability of it coming from θ_0 is negligible.

Setting	Upper Bound	Lower Bound
Fixed, known $\alpha, \theta_0, \theta_1$	$\frac{\log(1/(\delta\alpha))}{\alpha\epsilon^2}$, Thm. 7	$\frac{\log(\log(1/\delta)/\alpha)}{\alpha\epsilon^2}$, Cor. 1
Adaptive, known $\alpha, \theta_0, \theta_1$	$\frac{1}{\epsilon^2}\left(\frac{1}{\alpha} + \log(\frac{1}{\delta})\right)$, [28, 98], Thm. 5	$\frac{1}{\alpha\epsilon^2}$, [98]
Est+Expl, unknown $\alpha, \theta_0, \theta_1$	Unconsidered†	$\left(\frac{1}{\alpha\epsilon^2}\right)^2 \log(\frac{1}{\delta})$, Cor. 2
Adaptive, unknown $\alpha, \theta_0, \theta_1$	$\frac{c\log\left(\frac{1}{\alpha\epsilon^2}\right)\log\left(\log\left(\frac{1}{\alpha\epsilon^2}\right)/\delta\right)}{\alpha\epsilon^2}$, Thm. 6	$\frac{1}{\alpha\epsilon^2}$, [98]

Table 6.1: Upper and lower bounds on the expected sample complexity of different δ-probably correct strategies. Fixed refers to the strategy of Corollary 1. This table assumes $\min\{\theta_0(1-\theta_0), \theta_1(1-\theta_1)\}$ is lower bounded by a constant (e.g. $\theta_0, \theta_1 \in [1/8, 7/8]$) and $\epsilon = \theta_1 - \theta_0$ is sufficiently small. Also note that the upper bounds apply to distributions supported on $[0, 1]$, not just coins. All results without bracketed citations were unknown prior to this work. † Due to the discouraging lower bound for any estimate-then-explore strategy, it is inadvisable to propose an algorithm.

6.5 Near optimal adaptive algorithm

In this section I propose an algorithm that has no prior knowledge of the parameters $\alpha, \theta_0, \theta_1$ yet yields an upper bound that matches the lower bound of Theorem 1 up to logarithmic factors. I assume that samples from heavy or light distributions are supported on $[0, 1]$, and that drawn samples are independent and unbiased estimators of the mean, i.e., $\mathbb{E}[X_{i,j}] = \mu_i$ for $\mu_i \in \{\theta_0, \theta_1\}$. All results can be easily extended to sub-Gaussian distributions.

Algorithm overview. The near-optimal algorithm has two pieces, an outer loop that repeatedly guesses values for α and ϵ (Algorithm 6.3), and an inner loop that, given one such guess, draws multiple distributions and attempts to identify a heavy one (Algorithm 6.2). The key contribution of the inner loop is that it will identify a heavy distribution with high probability if it is given an underestimate of α and ϵ, and will output null with high probability if it is given an overestimate. As a result, the outer loop can simply pass decreasing estimates of α_0 and ϵ_0 to the inner loop until a heavy distribution is found. The main contribution of the outer loop is in the choice of the estimates α_0 and ϵ_0. The algorithm identifies 'landmarks' that balance decreasing α_0 against decreasing ϵ_0, shrinking them both simultaneously. As a result, it arrives at a low enough estimate very quickly.

As written, the algorithm describes how to find only a single heavy distribution, so it would require some minor modifications to run on a pool of crowd workers. To identify c fast workers at once, it suffices to run c instances of Algorithm 6.3 in parallel. Since each instance of the algorithm only considers a single worker at a time, the worker pool will always have c workers in it, as desired. In addition, the algorithm only describes how to identify a fast worker, not how to maintain fast workers over time. Maintaining the pool is a concern if workers abandon the pool, or if their speeds change over time. Again, this

only requires minor modifications to the algorithm. If a worker abandons the pool, a new worker can be hired to replace them (e.g., using the Pool Stability technique of Chapter 5), and the algorithm can continue running on that worker. Similarly, if workers' mean speeds display empirical drift, their work history can be periodically erased and the algorithm can treat them as a previously unseen worker.

Now, I state the theorems describing the complexity of the algorithm. Consider Algorithm 6.2, an SPRT-like procedure [128] for finding a heavy distribution given δ and lower bounds on α and $\epsilon = \theta_1 - \theta_0$. It improves upon prior work by supporting arbitrary distributions on $[0,1]$ and requires only bounds on α, ϵ.

Algorithm 6.2 Adaptive strategy for heavy distribution identification with inputs $\alpha_0, \epsilon_0, \delta$

Given $\delta \in (0, 1/4), \alpha_0 \in (0, 1/2), \epsilon_0 \in (0, 1)$.
Initialize $n = \lceil 2\log(9)/\alpha_0 \rceil, m = \lceil 64\epsilon_0^{-2} \log(14n/\delta) \rceil, A = -8\epsilon_0^{-1} \log(21)$,
  $B = 8\epsilon_0^{-1} \log(14n/\delta), k_1 = 5, k_2 = \lceil 8\epsilon_0^{-2} \log(2k_1/\min\{\delta/8, m^{-1}\epsilon_0^{-2}\}) \rceil$.
Draw k_1 distributions and sample them each k_2 times.
Estimate $\widehat{\theta}_0 = \min_{i=1,\ldots,k_1} \widehat{\mu}_{i,k_2}, \hat{\gamma} = \widehat{\theta}_0 + \epsilon_0/2$.
Repeat for $i = 1, \ldots, n$:
  Draw distribution i.
  Repeat for $j = 1, \ldots, m$:
    Sample distribution i and observe $X_{i,j}$.
    If $\sum_{k=1}^{j}(X_{i,k} - \hat{\gamma}) > B$:
      Declare distribution i to be heavy and **Output** distribution i.
    Else if $\sum_{k=1}^{j}(X_{i,k} - \hat{\gamma}) < A$:
      break.
Output `null`.

Theorem 5 *If Algorithm 6.2 is run with $\delta \in (0, 1/4), \alpha_0 \in (0, 1/2), \epsilon_0 \in (0, 1)$, then the expected number of total samples taken by the algorithm is no more than*

$$\frac{c'\alpha \log(1/\alpha_0) + c'' \log\left(\frac{1}{\delta}\right)}{\alpha_0 \epsilon_0^2} \tag{6.3}$$

for some absolute constants c',c'', and all of the following hold: 1) with probability at least $1-\delta$, a light distribution is not returned, 2) if $\epsilon_0 \leq \theta_1 - \theta_0$ and $\alpha_0 \leq \alpha$, then with probability $\frac{4}{5}$ a heavy distribution is returned, and 3) the procedure takes no more than $\frac{c\log(1/(\alpha_0\delta))}{\alpha_0\epsilon_0^2}$ total samples.

The second claim of the theorem holds only with constant probability (versus with probability $1-\delta$) since the probability of observing a heavy distribution among the $n = \lceil 2\log(4)/\alpha_0 \rceil$ distributions only occurs with constant probability. One can show that if the outer loop of algorithm is allowed to run indefinitely (with m and n defined as is), $\epsilon_0 = \theta_1 - \theta_0$,

$\alpha_0 = \alpha$, and $\widehat{\theta}_0 = \theta_0$, then a heavy coin is returned with probability at least $1-\delta$ and the expected number of samples is bounded by (6.3). If a tight lower bound is known on either $\epsilon = \theta_1 - \theta_0$ or α, there is only one parameter that is unknown and the "doubling trick", along with Theorem 5, can be used to identify a heavy coin with just $\frac{\log(\log(\epsilon^{-2})/\delta)}{\alpha\epsilon^2}$ and $\frac{\log(\log(\alpha^{-1})/\delta)}{\alpha\epsilon^2}$ samples, respectively (see Section 6.6.2).

Now consider Algorithm 6.3 that assumes no prior knowledge of $\alpha, \theta_0, \theta_1$, the first result for this setting that we are aware of. We remark that while the placing of "landmarks" (α_k, ϵ_k) throughout the search space as is done in Algorithm 6.3 appears elementary in hindsight, it is surprising that so few can cover this two dimensional space since one has to balance the exploration of α and ϵ. We believe similar a similar approach may be generalized for more generic infinite armed bandit problems.

Algorithm 6.3 Adaptive strategy for heavy distribution identification with unknown parameters

---

Given $\delta > 0$.
Initialize $\ell = 1$, heavy distribution $h =$ `null`.
Repeat until h is not `null`:
  Set $\gamma_\ell = 2^\ell, \delta_\ell = \delta/(2\ell^3)$
  Repeat for $k = 0, \ldots, \ell$:
    Set $\alpha_k = \frac{2^k}{\gamma_\ell}, \epsilon_k = \sqrt{\frac{1}{2\alpha_k\gamma_\ell}}$
    Run Algorithm 6.2 with $\alpha_0 = \alpha_k, \epsilon_0 = \epsilon_k, \delta = \delta_\ell$ and **Set** h to its output.
    If h is not `null` **break**
  Set $\ell = \ell + 1$
Output h

Theorem 6 (Unknown $\alpha, \theta_0, \theta_1$) *Fix $\delta \in (0,1)$. If Algorithm 6.3 is run with δ then with probability at least $1-\delta$ a heavy distribution is returned and the expected number of total samples taken is bounded by*

$$c\frac{\log_2(\frac{1}{\alpha\epsilon^2})}{\alpha\epsilon^2}(\alpha\log_2(\tfrac{1}{\epsilon^2}) + \log(\log_2(\tfrac{1}{\alpha\epsilon^2})) + \log(1/\delta))$$

for an absolute constant c.

6.6 Other Upper Bounds

The previous section presented the main contribution of this chapter, Algorithm 6.3, which is adaptive to unknown $\theta_0, \theta_1, \alpha$. In this section, I describe efficient algorithms for solving the most biased coin problem for the fixed sample size setting and settings where partial, but not all, information is known.

6.6.1 Fixed sample size strategy

Consider a fixed sample strategy that knows $\alpha, \theta_0, \theta_1$ and flips each coin exactly m times until it declares a coin as heavy (the algorithm can use the parameters to choose m). Note the result below is within a $\log(1/\delta)$ factor of the lower bound proved in Corollary 1 in general and is tight when $\alpha \leq \delta$.

Theorem 7 (Fixed sample size, known $\alpha, \theta_0, \theta_1$) *Fix $\delta \in (0, 1/4)$ and set $\widehat{n} = \left\lceil \frac{1}{\alpha} \log(\frac{2}{\delta}) \right\rceil$ and $m = \left\lceil \frac{2\log(4\widehat{n}/\delta)}{(\theta_1-\theta_0)^2} \right\rceil$. There exists a fixed sample size strategy with stopping time $N_m \leq \widehat{n}$ that is δ-probably correct and satisfies*

$$\mathbb{E}[mN_m] \leq 3\frac{\log(1/\alpha)+\log(12\log(6/\delta)/\delta)}{\alpha(\theta_1-\theta_0)^2} \leq 12\frac{\log(\frac{2}{\delta\alpha})}{\alpha(\theta_1-\theta_0)^2}.$$

6.6.2 Either α or ϵ is unknown, but not both

Algorithm 6.4 Algorithm for unknown θ_1, θ_0.

Given $\delta \in (0, 1), \alpha \in (0, 1/2)$.
Initialize $k = 1$
While Algorithm 6.2 run with inputs $\delta/(2k^2)$, $\alpha_0 = \alpha, \epsilon_0 = 2^{-k}$ returns `null`:
  Set $k = k + 1$.
Output distribution k.

Algorithm 6.5 Algorithm for unknown α.

Given $\delta \in (0, 1), \epsilon \in (0, 1]$.
Initialize $k = 1$
While Algorithm 6.2 run with inputs $\delta/(2k^2)$, $\alpha_0 = 2^{-k}, \epsilon_0 = \epsilon$ returns `null`:
  Set $k = k + 1$.
Output distribution k.

First we consider the case when α is known but a lower bound on $\theta_1 - \theta_0$ is not, and then opposite.

Theorem 8 (Known α, unknown θ_0, θ_1) *Fix $\delta \in (0, 1)$. If Algorithm 6.4 is run with δ, α then with probability at least $1 - \delta$ a heavy distribution is returned and the expected number of total samples taken is no more than*

$$\frac{c\log\left(\log\left(\frac{1}{(\theta_1-\theta_0)^2}\right)/\delta\right)}{\alpha(\theta_1-\theta_0)^2}.$$

for an absolute constant c.

Theorem 9 (Unknown α, known θ_0, θ_1) *Fix $\delta \in (0, 1)$. If Algorithm 6.5 is run with $\delta, \theta_1 - \theta_0$ then with probability at least $1 - \delta$ a heavy distribution is returned and the the expected number of total samples taken is no more than*

$$\frac{c\log\left(\log\left(\frac{1}{\alpha}\right)/\delta\right)}{\alpha(\theta_1-\theta_0)^2}$$

for an absolute constant c.

6.7 Conclusion

While all prior works have required at least partial knowledge of $\alpha, \theta_0, \theta_1$ to solve the most biased coin problem, the algorithm presented in this chapter (Algorithm 6.3) requires no knowledge of these parameters yet obtains a near-optimal sample complexity. In addition, I have proved lower bounds on the sample complexity of detecting the presence of a mixture distribution when the parameters are known or unknown. This has consequences for any estimate-then-explore strategy, an approach previously proposed for infinite-armed bandit problems. Extending this adaptive algorithm to arbitrary arm reservoir distributions is of significant interest. A successful algorithm in this vein could have a significant impact on how researchers think about sequential decision processes in both finite and uncountable action spaces.

In the broader context of this thesis, the analysis of Algorithm 6.3 provides provable guarantees on the rate at which a crowd system can converge on a fast pool of workers. Though the assumption that workers are either 'fast' or 'slow' is somewhat restrictive, the analysis techniques used here are an important step in the development of algorithms that model crowd behavior. The success of the adaptive approach to identifying fast workers and the disappointing lower bounds on estimate-then-explore strategies suggest that simply observing worker speeds and thresholding (as done by the system presented in Chapter 4) might be highly inefficient compared to the algorithms presented here.

Together, Chapters 3-6 describe the theory, design, and implementation of performant crowd-powered systems for data analysis. In the next and final chapter, I summarize the contributions of this thesis and conclude with thoughts about future opportunities for research in this space.

Chapter 7

Future Work and Conclusions

In this thesis, I have taken a comprehensive approach to the design, theory, and implementation of performant human-powered systems for data analysis. I have demonstrated how to build crowd systems that are fast enough to integrate into automated workflows, derived novel algorithms for identifying fast workers quickly, and described techniques to support many user applications running simultaneously on the same crowd. Throughout, I have leveraged existing work in database optimization, statistical learning theory, and cloud computing to demonstrate that approaching crowds of human workers with techniques from distributed computing can lead to significant improvement in performance, provided that human behavior is modeled as a part of the system. In this chapter, I conclude by summarizing the contributions of this work and describing the challenges that remain for future research.

7.1 Summary of Contributions

In Chapter 3, I described Wisteria, a system for interactive data cleaning that allows users to rapidly iterate on cleaning plans that involve human or automated operators. In addition to a novel architecture that treats sampling and crowdsourcing as first-class operators, Wisteria relies on consistent, low-latency responses from the crowd in order to provide a coherent user experience. This system therefore serves as an important motivation for reliable, performant crowd systems for data analytics.

In Chapter 4, I then described CLAMShell, a system designed to meet precisely those goals. CLAMShell tackles crowd latency at all of its sources in human-in-the-loop systems, including task latency, batch latency, and full-run latency. It uses straggler mitigation, pool maintenance, and hybrid active-and-passive learning to reduce the time it takes to get answers from the crowd from minutes or hours to seconds. My evaluation of the system on over 1,000 workers running nearly 250,000 tasks demonstrates that these techniques can reduce data processing latency by up to 8×, and reduce its variance by 150×. AMPCrowd, an open-source implementation of CLAMShell, has already been adopted to power solutions

in industry and novel research.

In Chapter 5, I described Cioppino, a multi-tenant crowd system that extends CLAMShell to serve many user applications in parallel. Driven by the insight that managing a multi-tenant crowd bears many similarities to distributed cluster management, Cioppino is architected around a queuing model of worker behavior. In addition, it leverages novel techniques to improve the performance and efficiency of the crowd, including control-theoretic auto-scaling of the crowd in response to changing application workloads, adaptive recruitment to compensate for worker abandonment, and proactive transfer of workers between applications to maximize utilization while respecting worker preferences. I evaluated Cioppino in simulation on a trace derived from workers hired by the CLAMShell system of Chapter 4, and found that costs were reduced by nearly 20×, while performance improved by 20%.

Finally, in Chapter 6, I turned my attention to the problem of identifying fast workers in an anonymous crowd. I connected this problem to the theoretical framework of the most biased coin problem, and identified an important relationship between that problem and statistical anomaly detection. I then proved lower bounds on the hardness of these problems, importantly demonstrating that estimate-then-explore strategies that try to estimate unknown distribution parameters and then use them to solve the most biased coin problem are quadratically more expensive than adaptive algorithms. Next, I presented a novel adaptive algorithm that finds fast crowd workers with no prior knowledge of the worker population distribution parameters, and proved near-optimal guarantees on its runtime. I additionally introduced near-optimal algorithms for doing so with only partial information.

7.2 Limitations

This work has demonstrated the practical viability of embedding low-latency crowdsourcing systems into automated data analysis workflows. As such, it focused primarily on the tradeoff between latency and cost rather than tradeoffs involving output quality. As I described in Chapter 2, much effort has been devoted to improving the quality of crowdsourced tasks, especially those involving sending each task to multiple workers and inferring task and worker quality from the results. Though I intentionally design my latency-oriented optimizations to be compatible with such quality control techniques, I do not evaluate the effectiveness of using them together.

Further, the experimental evaluation in this thesis performed on crowd workers was limited to workers hired on Amazon's Mechanical Turk. Though MTurk is a good choice for experimental evaluation due to its adoption by the academic community and its large worker population, it represents only a single crowd, and it could be argued that other crowds might display unaccounted-for behaviors that might impact the performance of the systems introduced in this thesis. The theoretical guarantees of Chapter 6 are made with no assumptions about the makeup of the crowd, and are therefore applicable to workers hired on any platform.

Finally, the systems in this thesis are most useful for applications running *microtasks*, where the desired work is short and simple and the outputs are well-defined. For example, in Chapter 4, the straggler mitigation technique assumes that it is feasible to send the same task to multiple workers and accept the first response, which would not be true of subjective or creative crowd tasks such as critiquing an essay, and the hybrid learning technique assumes task outputs can be used as labels for a machine learning training set. In general, approaches to improving the performance of heterogeneous macrotasks cannot take advantage of analogies to distributed computing, as do the techniques I have presented in this thesis.

7.3 Future Work

The field of crowd-powered systems is still in its infancy, and optimizations related to performance have only received attention in the past few years. As a result, there is ample opportunity to extend the ideas in this thesis to future research. In this section, I highlight a few of the more interesting potential directions.

Pushing individual techniques further. Because this thesis brings ideas and theory from many related areas to bear on crowd-powered systems, it is possible to leverage even more sophisticated ideas from those areas to improve the performance of crowd systems. For example, the hybrid learning technique described in Chapter 4 uses uncertainty sampling based on SVM margin distance to pick batches of points to be labeled by the crowd, an approach that has been widely adopted due to its simplicity and practical performance. However, the active learning community has developed theory for other algorithms such as query-by-committee that could be extended to fit the large-batch setting of crowd labeling. Similarly, the pool elasticity technique of Chapter 5 uses simple PID controllers to demonstrate that control theory can indeed be used to manage pools of crowd workers. However, the cloud autoscaling and control-theory literature takes advantage of more sophisticated approaches such as feed-forward control and time-series based regression analysis which could proactively scale the crowd to meet predicted workload changes.

Relaxing algorithmic assumptions. The algorithms introduced in Chapter 6 made a simplifying assumption that workers were either ‘fast’ workers with a high mean task processing rate or ‘slow’ workers with a low mean task processing rate. Future work could relax that assumption and look at the general case of worker means being drawn from an arbitrary distribution. The novel use of ‘landmarks’ to explore the parameter space described in this thesis could make this harder setting more amenable to analysis. Further, though this thesis proved guarantees on the number of tasks which must be completed to identify fast workers, the algorithms were not empirically evaluated on a live crowd. Future work could experimentally validate the theoretical guarantees.

Optimizing system models. The systems in this thesis each represent a single point on the spectrum between systems that treat humans as programming abstractions and those that model human behavior exhaustively. Future work in the design of performant crowd systems could investigate that spectrum itself. Identifying when it is necessary to model specific human behaviors in order to guarantee system performance would allow system designers to adaptively tune systems to the specific characteristics of the problem domain and the crowd population. Such meta-analysis and dynamic generation of system models could make designing performant crowd systems much easier for system users who do not have deep expertise in the area.

Designing tools for crowd system administration. Most real-world systems for automated data analysis are designed with companion tools for monitoring jobs, understanding performance bottlenecks, and debugging errors. Designing similar tools for the crowd is important for system designers, but comes with a unique set of challenges. For example, estimating overall job completion is challenging because of the high variance of crowd responses, and replaying tasks for debugging purposes is non-deterministic, and might incur unexpected cost and latency. The techniques introduced in this thesis to reduce the variance of data processing times are an important step towards meeting these challenges, but work remains to make reliable tools that support users of performant crowdsourcing systems.

7.4 Final Remarks

As large-scale data analysis applications have become increasingly ubiquitous, their reliance on human workers to label, clean, and extract data is growing in turn. In order to prevent human-powered steps from becoming the bottleneck in data analysis workflows, there is a clear need for performant crowd systems. In this thesis, I have described novel techniques for designing crowd systems that support multiple applications processing data at near-machine latencies, and presented provable algorithms for making them efficient. Many of the systems described in this thesis have been released as open-source software that is already being adopted by others in industry and academia.

In 1960, J.C.R. Licklider wrote about a future containing 'Man-Computer Symbiosis', a partnership between human and machine that "will think as no human brain has ever thought and process data in a way not approached by the information-handling machines we know today" [94]. More than 50 years later, crowd-powered computer systems are in some sense the fulfillment of his idea, thinking with the power of many human brains, augmented by algorithms, and processing data at the speed of powerful clusters of computers. Symbiosis, however, is just one model of human-computer interaction, and in a climate of growing concern over the role of automation in the workforce, we are at a critical juncture in the future of computing. Will machine learning become so advanced that it obviates the need for human expertise to extract insights from data? Or will we as a society identify a way

forward for people to collaborate with algorithms, even as those algorithms become more advanced?

In my opinion, we have little to fear. Not only is automated data analysis still far from being able to extract insights of human caliber consistently, but automated insights are of little use without the context, intent, and attention of the people who generate and consume them. Approaches that use algorithms to augment and synthesize the information that humans use to make decisions, while simultaneously using human expertise to improve those algorithms represents a promising path forwards. I firmly believe that in another 50 years, we will still have systems that allow humans and machines to work in concert, each contributing where they are most able.

Bibliography

[1] J. Acharya, C. Daskalakis, and G. C. Kamath. Optimal testing for properties of distributions. In *Advances in Neural Information Processing Systems*, pages 3577–3598, 2015.

[2] A. Agarwal, B. Xie, I. Vovsha, O. Rambow, and R. Passonneau. Sentiment analysis of Twitter data. *LASM*, 2011.

[3] D. Agarwal. Detecting anomalies in cross-classified streams: a bayesian approach. *Knowledge and Information Systems*, 11(1):29–44, 2006.

[4] D. Alba. The Hidden Laborers Training AI to Keep Ads Off Hateful YouTube Videos. https://www.wired.com/2017/04/zerochaos-google-ads-quality-raters/, April 2017.

[5] A. Ali-Eldin, J. Tordsson, and E. Elmroth. An adaptive hybrid elasticity controller for cloud infrastructures. In *NOMS*, pages 204–212. IEEE, 2012.

[6] Amazon. Amazon auto scaling service. http://aws.amazon.com/autoscaling/, 2016.

[7] Amazon. Amazon Mechanical Turk API Reference. http://docs.aws.amazon.com/AWSMechTurk/latest/AWSMturkAPI/Welcome.html, 2017.

[8] American National Standards Institute. *American national standard for information systems: database language — SQL: ANSI X3.135-1992*. American National Standards Institute, 1992.

[9] AMPCrowd. https://amplab.github.io/ampcrowd.

[10] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective Straggler Mitigation: Attack of the Clones. *NSDI*, 2013.

[11] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the Outliers in Map-Reduce Clusters using Mantri. *OSDI*, 2010.

[12] C. Aniszczyk. Crowdsourced data analysis with clockwork raven. https://blog.twitter.com/engineering/en_us/a/2012/crowdsourced-data-analysis-with-clockwork-raven.html, 2012.

[13] Apache falcon. http://falcon.apache.org.

[14] A. Arasu and H. Garcia-Molina. Extracting structured data from web pages. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 337–348, 2003.

[15] T. S. Behrend, D. J. Sharek, A. W. Meade, and E. N. Wiebe. The viability of crowdsourcing for survey research. *Behavior Research Methods*, 43(3):800, Mar 2011.

[16] M. S. Bernstein, J. Brandt, R. C. Miller, and D. R. Karger. Crowds in two seconds: enabling realtime crowd-powered interfaces. *UIST*, 2011.

[17] M. S. Bernstein, D. R. Karger, R. C. Miller, and J. Brandt. Analytic Methods for Optimizing Realtime Crowdsourcing. *Collective Intelligence*, 2012.

[18] M. S. Bernstein, G. Little, R. C. Miller, B. Hartmann, M. S. Ackerman, D. R. Karger, D. Crowell, and K. Panovich. Soylent: a word processor with a crowd inside. *UIST*, 2010.

[19] D. A. Berry, R. W. Chen, A. Zame, D. C. Heath, and L. A. Shepp. Bandit problems with infinitely many arms. *Ann. Statist.*, 25(5):2103–2116, 10 1997.

[20] J. P. Bigham, C. Jayant, H. Ji, G. Little, A. Miller, R. C. Miller, A. Tatarowicz, B. White, S. White, and T. Yeh. *VizWiz: nearly real-time answers to visual questions.* UIST, 2010.

[21] T. Bonald and A. Proutiere. Two-target algorithms for infinite-armed bandits with bernoulli rewards. In C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 2184–2192. Curran Associates, Inc., 2013.

[22] M. J. Cafarella, A. Halevy, D. Z. Wang, E. Wu, and Y. Zhang. Webtables: Exploring the power of tables on the web. *Proc. VLDB Endow.*, 1(1):538–549, Aug. 2008.

[23] C. Callison-Burch. Fast, cheap, and creative: evaluating translation quality using Amazon's Mechanical Turk. *EMNLP*, 2009.

[24] C. C. Cao, Z. Liu, L. Chen, and H. V. Jagadish. Tuning crowdsourced human computation. *CoRR*, abs/1610.04429, 2016.

[25] A. Carpentier and M. Valko. Simple regret for infinitely many armed bandits. *arXiv preprint arXiv:1505.04627*, 2015.

[26] Ç. Demiralp, M. S. Bernstein, and J. Heer. Learning perceptual kernels for visualization design. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):1933–1942, Dec 2014.

[27] S. Chakraborty, V. Balasubramanian, and S. Panchanathan. Adaptive Batch Mode Active Learning. *Trans. Neural Netw. Learning Sys.*, 2015.

[28] K. Chandrasekaran and R. Karp. Finding a most biased coin with fewest flips. In *Proceedings of The 27th Conference on Learning Theory*, pages 394–407, 2014.

[29] K. Chandrasekaran and R. M. Karp. Finding the most biased coin with fewest flips. *CoRR*, abs/1202.3639, 2012.

[30] K. Chen, A. Kannan, Y. Yano, J. M. Hellerstein, and T. S. Parikh. Shreddr: Pipelined paper digitization for low-resource organizations. In *Proceedings of the 2Nd ACM Symposium on Computing for Development*, ACM DEV '12, pages 3:1–3:10, 2012.

[31] K.-T. Chen, C.-C. Wu, Y.-C. Chang, and C.-L. Lei. A crowdsourceable qoe evaluation framework for multimedia content. In *Proceedings of the 17th ACM International Conference on Multimedia*, MM '09, pages 491–500. ACM, 2009.

[32] Z. Chen and M. Cafarella. Integrating spreadsheet data via accurate and low-effort extraction. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1126–1135. ACM, 2014.

[33] J. Cheng and M. S. Bernstein. Flock: Hybrid crowd-machine learning classifiers. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*, CSCW '15, pages 600–611, 2015.

[34] X. Chu, J. Morcos, I. F. Ilyas, M. Ouzzani, P. Papotti, N. Tang, and Y. Ye. Katara: A data cleaning system powered by knowledge bases and crowdsourcing. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1247–1261, 2015.

[35] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[36] D. A. Cohn, Z. Ghahramani, and M. I. Jordan. Active Learning with Statistical Models. *JAIR*, 1996.

[37] D. Crankshaw, P. Bailis, J. E. Gonzalez, H. Li, Z. Zhang, M. J. Franklin, A. Ghodsi, and M. I. Jordan. The missing piece in complex analytics: Low latency, scalable model management and serving with velox. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*, 2015.

[38] CrowdCrafting. `https://crowdcrafting.org`.

[39] CrowdFlower. `https://crowdflower.com`.

[40] M. Dallachiesa, A. Ebaid, A. Eldawy, A. K. Elmagarmid, I. F. Ilyas, M. Ouzzani, and N. Tang. Nadeef: a commodity data cleaning system. In *SIGMOD Conference*, pages 541–552, 2013.

[41] A. Das Sarma, A. Parameswaran, H. Garcia-Molina, and A. Halevy. Crowd-powered find algorithms. *ICDE*, 2014.

[42] A. P. Dawid and A. M. Skene. Maximum likelihood estimation of observer error-rates using the em algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):20–28, 1979.

[43] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 2008.

[44] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 127–144, New York, NY, USA, 2014. ACM.

[45] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

[46] J. Deng, J. Krause, and L. Fei-Fei. Fine-grained crowdsourcing for fine-grained recognition. In *Proceedings of the 2013 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '13, pages 580–587, 2013.

[47] S. Dent. Google's ai gets human help to avoid offensive search results. `https://www.engadget.com/2017/03/15/googles-ai-gets-human-help-to-avoid-offensive-search-results/`, May 2017.

[48] V. Dhar. Data science and prediction. *Commun. ACM*, 56(12):64–73, Dec. 2013.

[49] D. E. Difallah, G. Demartini, and P. Cudré-Mauroux. Scheduling human intelligence tasks in multi-tenant crowd-powered systems. In *Proceedings of the 25th International Conference on World Wide Web*, WWW '16, pages 855–865. International World Wide Web Conferences Steering Committee, 2016.

[50] A. Doan, R. Ramakrishnan, and A. Y. Halevy. Crowdsourcing systems on the world-wide web. *Commun. ACM*, 54(4):86–96, Apr. 2011.

[51] E. Eskin. Anomaly detection over noisy data using learned probability distributions. In *Proceedings of the Seventeenth International Conference on Machine Learning*, ICML '00, pages 255–262, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[52] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. CrowdDB: answering queries with crowdsourcing. *SIGMOD*, 2011.

[53] Y. Freund and Y. Mansour. Estimating a mixture of two product distributions. In *Proceedings of the twelfth annual conference on Computational learning theory*, pages 53–62. ACM, 1999.

[54] GalaxyZoo. `https://galaxyzoo.org`.

[55] Y. Gao and A. Parameswaran. Finish them!: Pricing algorithms for human computation. *Proc. VLDB*, 7(14):1965–1976, Oct. 2014.

[56] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 323–336, Berkeley, CA, USA, 2011. USENIX Association.

[57] B. V. Gnedenko and I. N. Kovalenko. *Introduction to Queueing Theory (2Nd Ed).* Birkhauser Boston Inc., Cambridge, MA, USA, 1989.

[58] C. Gokhale, S. Das, A. Doan, J. F. Naughton, N. Rampalli, J. Shavlik, and X. Zhu. Corleone: hands-off crowdsourcing for entity matching. *SIGMOD*, 2014.

[59] Google. Autoscaling groups of instances. `https://cloud.google.com/compute/docs/autoscaler/`, 2017.

[60] D. Grimaldi, V. Persico, A. Pescape, A. Salvi, and S. Santini. A feedback-control approach for resource management in public clouds. In *2015 IEEE Global Communications Conference (GLOBECOM)*, pages 1–7, Dec 2015.

[61] P. Guo. Data Science Workflow: Overview and Challenge. `https://cacm.acm.org/blogs/blog-cacm/169199-data-science-workflow-overview-and-challenges/fulltext`, 2013.

[62] I. Guyon. Design of experiments for the NIPS 2003 variable selection benchmark, 2003.

[63] D. Haas, J. Ansel, L. Gu, and A. Marcus. Argonaut: Macrotask crowdsourcing for complex data processing. *Proc. VLDB*, 8(12):1642–1653, Aug. 2015.

[64] D. Haas and M. J. Franklin. Cioppino: Multi-Tenant Crowd Management. In *Proceedings of the fifth AAAI Conference on Human Computation and Crowdsourcing (HCOMP)*, 2017.

[65] D. Haas, S. Krishnan, J. Wang, M. J. Franklin, and E. Wu. Wisteria: Nurturing Scalable Data Cleaning Infrastructure. *VLDB*, 2015.

[66] D. Haas, J. Wang, E. Wu, and M. J. Franklin. Clamshell: Speeding up crowds for low-latency data labeling. *Proc. VLDB*, 9(4):372–383, Dec. 2015.

[67] J. Hackman and G. R. Oldham. Motivation through the design of work: test of a theory. *Organizational Behavior and Human Performance*, 16(2):250 – 279, 1976.

[68] Hadoop. `http://hadoop.apache.org/`.

[69] M. Hardt and E. Price. Sharp bounds for learning a mixture of two gaussians. *ArXiv e-prints*, 1404, 2014.

[70] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, SIGMOD '97, pages 171–182, 1997.

[71] K. Ikeda, A. Morishima, H. Rahman, S. B. Roy, S. Thirumuruganathan, S. Amer-Yahia, and G. Das. Collaborative crowdsourcing with crowd4u. *Proc. VLDB Endow.*, 9(13):1497–1500, Sept. 2016.

[72] Informatica. `https://www.informatica.com`.

[73] P. Ipeirotis and J. Horton. Visualizations of the oDesk "oConomy": Exploring Our World of Work. `https://www.upwork.com/blog/2012/07/visualizations-of-odesk-oconomy/`, 2012.

[74] P. G. Ipeirotis. Analyzing the Amazon Mechanical Turk marketplace. *ACM Crossroads*, 2010.

[75] P. G. Ipeirotis, F. Provost, and J. Wang. Quality management on Amazon Mechanical Turk. *SIGKDD*, 2010.

[76] W. Iqbal, M. N. Dailey, and D. Carrera. Unsupervised learning of dynamic resource provisioning policies for cloud-hosted multitier web applications. *IEEE Systems Journal*, 10(4):1435–1446, Dec 2016.

[77] K. Jamieson and R. Nowak. Best-arm identification algorithms for multi-armed bandits in the fixed confidence setting. In *Information Sciences and Systems (CISS)*, pages 1–6. IEEE, 2014.

[78] K. G. Jamieson, D. Haas, and B. Recht. The power of adaptivity in identifying statistical alternatives. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 775–783, 2016.

[79] J. Jiang, J. Lu, G. Zhang, and G. Long. Optimal cloud resource auto-scaling for web applications. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, 2013.

[80] M. Joglekar, H. Garcia-Molina, and A. Parameswaran. *Comprehensive and reliable crowd assessment algorithms*, volume 2015-May, pages 195–206. IEEE Computer Society, 5 2015.

[81] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Enterprise data analysis and visualization: An interview study. *VAST*, 2012.

[82] S. Kandel, A. Paepcke, J. M. Hellerstein, and J. Heer. Wrangler: interactive visual specification of data transformation scripts. *CHI*, 2011.

[83] D. R. Karger, S. Oh, and D. Shah. Iterative Learning for Reliable Crowdsourcing Systems. *Advances in neural information processing systems (NIPS)*, 2011.

[84] A. Kittur, E. H. Chi, and B. Suh. Crowdsourcing user studies with Mechanical Turk. *CHI*, 2008.

[85] A. Kittur, B. Smus, S. Khamkar, and R. E. Kraut. Crowdforge: Crowdsourcing complex work. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, UIST '11, pages 43–52, New York, NY, USA, 2011. ACM.

[86] R. A. Krishna, K. Hata, S. Chen, J. Kravitz, D. A. Shamma, L. Fei-Fei, and M. S. Bernstein. Embracing error to enable rapid crowdsourcing. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, pages 3167–3179, 2016.

[87] S. Krishnan, J. Wang, M. J. Franklin, K. Goldberg, and T. Kraska. Stale view cleaning: Getting fresh answers from stale materialized views. *Proc. VLDB*, 8(12), 2015.

[88] A. Krizhevsky. Learning multiple layers of features from tiny images, 2009.

[89] G. P. Krueger. Sustained work, fatigue, sleep loss and performance: A review of the issues. *Work & Stress*, 2007.

[90] A. Kulkarni, P. Narula, D. Rolnitzky, and N. Kontny. Wish: Amplifying creative ability with expert crowds. In *Proceedings of the Seconf AAAI Conference on Human Computation and Crowdsourcing, HCOMP 2014, November 2-4, 2014, Pittsburgh, Pennsylvania, USA*, 2014.

[91] W. S. Lasecki, Y. C. Song, H. Kautz, and J. P. Bigham. Real-time crowd labeling for deployable activity recognition. In *Proceedings of the 2013 Conference on Computer Supported Cooperative Work*, CSCW '13, pages 1203–1212, 2013.

[92] LeadGenius. `https://www.leadgenius.com/`.

[93] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[94] J. C. R. Licklider. Man-Computer Symbiosis. *IRE Transactions on Human Factors in Electronics*, HFE-1:4–11, March 1960.

[95] G. Little. Turkit: Tools for iterative tasks on mechanical turk. In *2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 252–253, Sept 2009.

[96] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *J. Grid Comput.*, 12(4):559–592, Dec. 2014.

[97] M. Lugo. The expectation of the maximum of exponentials. `http://www.stat.berkeley.edu/~mlugo/stat134-f11/exponential-maximum.pdf`, October 2011.

[98] M. L. Malloy, G. Tang, and R. D. Nowak. Quickest search for a rare distribution. In *Information Sciences and Systems (CISS)*, pages 1–6. IEEE, 2012.

[99] A. Marcus, D. Karger, S. Madden, R. Miller, and S. Oh. Counting with the crowd. *VLDB*, 2012.

[100] A. Marcus and A. Parameswaran. Crowdsourced data management: Industry and academic perspectives. *Foundations and Trends in Databases*, 2015.

[101] A. Marcus, E. Wu, D. Karger, S. Madden, and R. Miller. Human-powered sorts and joins. *VLDB*, 2011.

[102] A. Marcus, E. Wu, D. R. Karger, S. Madden, and R. C. Miller. Crowdsourced Databases: Query Processing with People. *CIDR*, 2011.

[103] C. Mayfield, J. Neville, and S. Prabhakar. Eracer: a database approach for statistical inference and data cleaning. In *SIGMOD*, 2010.

[104] B. Mozafari, P. Sarkar, M. J. Franklin, M. Jordan, and S. Madden. Scaling up crowdsourcing to very large datasets: a case for active learning. *VLDB*, 2014.

[105] Amazon Mechanical Turk. `https://www.mturk.com/`.

[106] P. Narula, P. Gutheim, D. Rolnitzky, A. Kulkarni, and B. Hartmann. Mobileworks: A mobile crowdsourcing platform for workers at the bottom of the pyramid. In *Proceedings of the 11th AAAI Conference on Human Computation*, AAAIWS'11-11, pages 121–123. AAAI Press, 2011.

[107] M. A. Netto, C. Cardonha, R. L. Cunha, and M. D. Assunçao. Evaluating auto-scaling strategies for cloud computing environments. In *Modelling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2014 IEEE 22nd International Symposium on*, pages 187–196. IEEE, 2014.

[108] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 69–84, New York, NY, USA, 2013. ACM.

[109] A. G. Parameswaran, H. Garcia-Molina, H. Park, N. Polyzotis, A. Ramesh, and J. Widom. Crowdscreen: Algorithms for filtering data with humans. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 361–372, New York, NY, USA, 2012. ACM.

[110] A. G. Parameswaran, H. Park, H. Garcia-Molina, N. Polyzotis, and J. Widom. Deco: declarative crowdsourcing. *CIKM*, 2012.

[111] A. G. Parameswaran, M. H. Teh, H. Garcia-Molina, and J. Widom. DataSift: An Expressive and Accurate Crowd-Powered Search Toolkit. *HCOMP*, 2013.

[112] H. Park and J. Widom. Crowdfill: Collecting structured data from the crowd. In *SIGMOD*, 2014.

[113] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[114] B. Peters. The Age of Big Data. `https://www.forbes.com/sites/bradpeters/2012/07/12/the-age-of-big-data`, July 2012.

[115] D. Pollard. Asymptopia. *Manuscript in progress. Available at http://www.stat.yale.edu/∼pollard*, 2000.

[116] C. Qu, R. N. Calheiros, and R. Buyya. Auto-scaling web applications in clouds: A taxonomy and survey. *CoRR*, abs/1609.09224, 2016.

[117] A. J. Quinn and B. B. Bederson. Human computation: a survey and taxonomy of a growing field. In *Proceedings of the International Conference on Human Factors in Computing Systems, CHI 2011, Vancouver, BC, Canada, May 7-12, 2011*, pages 1403–1412, 2011.

[118] A. Ramesh, A. Parameswaran, H. Garcia-Molina, and N. Polyzotis. Identifying Reliable Workers Swiftly. Technical report, Stanford University, 2012.

[119] D. Retelny, S. Robaszkiewicz, A. To, W. S. Lasecki, J. Patel, N. Rahmati, T. Doshi, M. Valentine, and M. S. Bernstein. Expert crowdsourcing with flash teams. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, UIST '14, pages 75–85, New York, NY, USA, 2014. ACM.

[120] J. Rogstadius, V. Kostakos, A. Kittur, B. Smus, J. Laredo, and M. Vukovic. An assessment of intrinsic and extrinsic motivation on task performance in crowdsourcing markets. In *ICWSM*, 2011.

[121] J. M. Rzeszotarski, E. Chi, P. Paritosh, and P. Dai. Inserting micro-breaks into crowdsourcing workflows. In *First AAAI Conference on Human Computation and Crowdsourcing*, 2013.

[122] J. M. Rzeszotarski and A. Kittur. Instrumenting the crowd: Using implicit behavioral measures to predict task performance. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, UIST '11, pages 13–22, New York, NY, USA, 2011. ACM.

[123] Samasource. `https://www.samasource.org/`.

[124] SampleClean. `http://sampleclean.org`.

[125] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, SIGMOD '79, pages 23–34, 1979.

[126] B. Settles. Active learning literature survey. Technical report, University of Wisconsin-Madison, 2010.

[127] V. S. Sheng, F. Provost, and P. G. Ipeirotis. Get another label? improving data quality and data mining using multiple, noisy labelers. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, pages 614–622, 2008.

[128] D. Siegmund. *Sequential analysis: tests and confidence intervals*. Springer Science & Business Media, 2013.

[129] R. Spira. Calculation of the gamma function by stirling's formula. *mathematics of computation*, pages 317–322, 1971.

[130] M. Stonebraker, D. Bruckner, I. F. Ilyas, G. Beskales, M. Cherniack, S. B. Zdonik, A. Pagan, and S. Xu. Data Curation at Scale: The Data Tamer System. *CIDR*, 2013.

[131] Talend. `https://www.talend.com/solutions/etl-analytics`.

[132] Tamr. `https://www.tamr.com/`.

[133] O. Tamuz, C. Liu, S. J. Belongie, O. Shamir, and A. Kalai. Adaptively learning the crowd kernel. In *Proceedings of the 28th International Conference on Machine Learning, ICML 2011, Bellevue, Washington, USA, June 28 - July 2, 2011*, pages 673–680, 2011.

[134] TaskUs. https://www.taskus.com//.

[135] G. Thatte, U. Mitra, and J. Heidemann. Parametric methods for anomaly detection in aggregate traffic. *IEEE/ACM Trans. Netw.*, 19(2):512–525, Apr. 2011.

[136] Trifacta. http://www.trifacta.com.

[137] B. Trushkowsky, T. Kraska, M. J. Franklin, and P. Sarkar. Crowdsourced enumeration queries. *ICDE*, 2013.

[138] UpWork. https://www.upwork.com/.

[139] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, Aug. 1990.

[140] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. J. Franklin, and I. Stoica. The power of choice in data-aware cluster scheduling. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 301–316, Berkeley, CA, USA, 2014. USENIX Association.

[141] R. Verborgh and M. De Wilde. *Using OpenRefine*. Packt Publishing Ltd, 2013.

[142] J. Wang, T. Kraska, M. J. Franklin, and J. Feng. CrowdER: Crowdsourcing Entity Resolution. *VLDB*, 2012.

[143] J. Wang, S. Krishnan, M. J. Franklin, K. Goldberg, T. Kraska, and T. Milo. A sample-and-clean framework for fast and accurate query processing on dirty data. *SIGMOD*, 2014.

[144] J. Wang, G. Li, T. Kraska, M. J. Franklin, and J. Feng. Leveraging transitive relations for crowdsourced joins. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 229–240, New York, NY, USA, 2013. ACM.

[145] Y. Wang, J. yves Audibert, and R. Munos. Algorithms for infinitely many-armed bandits. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, *Advances in Neural Information Processing Systems 21*, pages 1729–1736. Curran Associates, Inc., 2009.

[146] E. Wu, L. Jiang, L. Xu, and A. Nandi. Graphical perception in animated bar charts. *CoRR*, abs/1604.00080, 2016.

[147] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. *NSDI*, 2012.

[148] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. *OSDI*, 2008.

[149] ZeroChaos. http://www.zerochaos.com/.

[150] M. Zuckerberg. Facebook post. https://www.facebook.com/zuck/posts/10103695315624661, May 2017.

Appendix A

Proofs of Chapter 6 Theorems

A.1 Proof of Theorem 2

Proof First, let N be the number of distributions considered at the stopping time T. Note that $T \geq N$. By assumption the procedure satisfies $\mathbb{P}_1(N \geq n| \cap_{i=1}^{n-1} \{\xi_i = 0\}) \geq 1-\delta$ for all $n \in \mathbb{N}$. And

$$\begin{aligned}\mathbb{P}_1(N \geq n) \geq \mathbb{P}_1(N \geq n, \cap_{i=1}^{n-1}\{\xi_i = 0\}) &= \mathbb{P}_1(N \geq n| \cap_{i=1}^{n-1} \{\xi_i = 0\})\mathbb{P}_1(\cap_{i=1}^{n-1}\{\xi_i = 0\}) \\ &\geq (1-\delta)(1-\alpha)^{n-1}\end{aligned}$$

Thus, $\mathbb{E}_1[N] = \sum_{n=1}^{\infty} \mathbb{P}_1(N \geq n) \geq (1-\delta)\sum_{n=1}^{\infty}(1-\alpha)^{n-1} = \frac{1-\delta}{\alpha}$ which results in the first argument of the max.

Applying Theorem 2.38 of [128] we have

$$\mathbb{E}_1[N]\chi^2\left(\mathbb{P}_1|\mathbb{P}_0\right) \overset{\text{Eqn. 6.2}}{\geq} \mathbb{E}_1[N]KL\left(\mathbb{P}_1|\mathbb{P}_0\right) \overset{\text{Thm. 2.38}}{\geq} \log(\tfrac{1}{P_0(N<\infty)}) \overset{\text{assumption}}{\geq} \log(\tfrac{1}{\delta}),$$

which results in the second argument of the max.

If $\widetilde{\Theta} = \{\theta_0\}$ then $\chi^2(\mathbb{P}_1|\mathbb{P}_0) = \chi^2((1-\alpha)f_{\theta_0} + \alpha f_{\theta_1}|f_{\theta_0})$ and

$$\chi^2((1-\alpha)f_{\theta_0} + \alpha f_{\theta_1}|f_{\theta_0}) = \int \frac{\left((1-\alpha)f_{\theta_0}(x) + \alpha f_{\theta_1}(x) - f_{\theta_0}(x)\right)^2}{f_{\theta_0}(x)}dx = \alpha^2\chi^2(f_{\theta_1}|f_{\theta_0})$$

Thus, $\mathbb{E}_1[N] \geq \frac{\log(\frac{1}{\delta})}{\alpha^2\chi^2(f_{\theta_1}|f_{\theta_0})}$ which results in the second part of the theorem. ■

A.2 Proof of Corollary 1

Proof For $k = 0, 1$ let g_{θ_k} be a Bernoulli distribution with parameter θ_k and let $f_{\theta_k} = g_{\theta_k} \otimes \cdots \otimes g_{\theta_k}$ be a product distribution composed of m g_{θ_k} distributions. Then

$$\chi^2(g_{\theta_1}|g_{\theta_0}) = \frac{(\theta_1-\theta_0)^2}{\theta_0(1-\theta_0)} \leq e^{\frac{(\theta_1-\theta_0)^2}{\theta_0(1-\theta_0)}} - 1$$

and

$$\chi^2(f_{\theta_1}|f_{\theta_0}) = \left(1+\chi^2(g_{\theta_1}|g_{\theta_0})\right)^m - 1 \le e^{m\frac{(\theta_1-\theta_0)^2}{\theta_0(1-\theta_0)}} - 1.$$

Moreover, $e^{m\frac{(\theta_1-\theta_0)^2}{\theta_0(1-\theta_0)}} - 1 \le m\frac{(\theta_1-\theta_0)^2}{\theta_0(1-\theta_0)}$ whenever $m \le \frac{\theta_0(1-\theta_0)}{2(\theta_1-\theta_0)^2}$ since $e^{x/2}-1 \le x$ for all $x \in [0,1]$. Applying Theorem 2 obtains

$$\mathbb{E}[N_m] \ge \max\left\{ \frac{1-\delta}{\alpha}, \frac{\log(\frac{1}{\delta})}{\alpha^2(e^{m\frac{(\theta_1-\theta_0)^2}{\theta_0(1-\theta_0)}}-1)} \right\} \ge \frac{\theta_0(1-\theta_0)\log(\frac{1}{\delta})}{m\alpha^2(\theta_1-\theta_0)^2}\mathbf{1}_{m\le\frac{\theta_0(1-\theta_0)}{2(\theta_1-\theta_0)^2}}\ .$$

The claimed result follows from loosening the integer constraint on m and minimizing the lower bound on $\mathbb{E}[N_m]$ multiplied by m. To perform the minimization, we note that the function $\max\{\frac{1-\delta}{\alpha}, 2\log(\frac{1}{\delta})/[\alpha^2(e^{m\frac{(\theta_1-\theta_0)^2}{\theta_0(1-\theta_0)}}-1)]\}$ reaches its minimum at the intersection of the two arguments and solve for m at that point. ■

A.3 Proof of Theorem 3

Proof I begin by restating the theorem with the problem dependent parameters explicitly defined.

Theorem 10 *Suppose f_θ for $\theta \in \Theta \subset \mathbb{R}$ is a single parameter exponential family so that $f_\theta(x) = h(x)\exp(\eta(\theta)x - b(\eta(\theta)))$ for some scalar functions h, b, η where η is strictly increasing. If $\mathbb{E}_\theta[X] = \int x f_\theta(x)dx$ then let $M_k(\theta) = \int (x-\mathbb{E}_\theta[X])^k f_\theta(x)dx$ denote the kth centered moment under distribution f_θ. Define*

$$\begin{aligned} \theta_* &= \eta^{-1}\big((1-\alpha)\eta(\theta_0) + \alpha\eta(\theta_1)\big) \\ \theta_- &= \eta^{-1}\big(\eta(\theta_0) - \alpha(\eta(\theta_1)-\eta(\theta_0))\big) \\ \theta_+ &= \eta^{-1}\big(\eta(\theta_1) + (1-\alpha)(\eta(\theta_1)-\eta(\theta_0))\big) \end{aligned}$$

and assume there exist finite κ, γ such that

$$\sup_{y\in[\theta_0,\theta_1]} b(2\eta(y)-\eta(\theta_*)) - [2b(\eta(y)) - b(\eta(\theta_*))] \le \kappa, \quad \textit{and} \quad \sup_{x\in[\dot b(\eta(\theta_-)),\dot b(\eta(\theta_+))]} \phi_x(\dot b^{-1}(x)) \le \gamma,$$

where $\phi_x(\eta(\theta)) = f_\theta(x)$. Then

$$\chi^2((1-\alpha)f_{\theta_0}(x) + \alpha f_{\theta_1}(x)|f_{\theta_*}(x)) \le c\left(\frac{1}{2}\alpha(1-\alpha)(\eta(\theta_1)-\eta(\theta_0))^2\right)^2$$

where if $\Delta = \dot b(\eta(\theta_+)) - \dot b(\eta(\theta_-))$

$$c = e^\kappa\left(\sup_{\theta\in[\theta_0,\theta_1]} M_2(\theta)^2\ \ (2+\gamma\Delta) + 8M_4(\theta_-) + 8M_4(\theta_+) + 16\Delta^4 + \tfrac{2}{5}\gamma\Delta^5\right).$$

Thus, if $\widetilde{\Theta} = \{\theta_*\}$ and N is the stopping time of any procedure that satisfies $\mathbb{P}_0(N < \infty) \leq \delta$ and $\mathbb{P}_1(\cup_{i=1}^N \{\xi_i = 1\}) \geq 1 - \delta$, then

$$\mathbb{E}_1[N] \geq \max\left\{ \frac{1-\delta}{\alpha}, \frac{\log(\frac{1}{\delta})}{c\left(\frac{1}{2}\alpha(1-\alpha)(\eta(\theta_1) - \eta(\theta_0))^2\right)^2} \right\}.$$

Proof Define $\phi_x(\eta) = h(x)\exp(\eta x - b(\eta))$. By the properties of scalar exponential families, note that $b'(\eta)$ and $b''(\eta) \geq 0$ represent the mean and variance of the distribution. We deduce that b' is monotonically increasing. Define $\eta_0 = \eta(\theta_0)$, $\eta_1 = \eta(\theta_1)$, and $\mu = (1-\alpha)\eta_0 + \alpha\eta_1$. Noting that

$$\chi^2((1-\alpha)\phi_x(\eta_0) + \alpha\phi_x(\eta_1) | \phi_x(\mu)) = \int \phi_x(\mu) \left(\frac{(1-\alpha)\phi_x(\eta_0) + \alpha\phi_x(\eta_1) - \phi_x(\mu)}{\phi_x(\mu)} \right)^2 dx,$$

I will use a technique that was used in [115] to approximate the divergence between a single Gaussian distribution and a mixture of them. Essentially, it involves taking the Taylor series of each $\phi_x(\cdot)$ centered at μ and bounding it. We have

$$\begin{aligned} \phi_x(\eta) &= h(x)\exp(\eta x - b(\eta)) \\ \phi_x'(\eta) &= (x - b'(\eta))\phi_x(\eta) \\ \phi_x''(\eta) &= (-b''(\eta) + (x - b'(\eta))^2)\phi_x(\eta) \end{aligned}$$

so that

$$\phi_x(y) = \phi_x(\mu)\left[1 + (x - b'(\mu))(y-\mu) + \tfrac{1}{2}(-b''(\mu) + (x - b'(\mu))^2)(y-\mu)^2 \dots\right].$$

Noting that $(\eta_0 - \mu) = -\alpha(\eta_1 - \eta_0)$, $(\eta_1 - \mu) = (1-\alpha)(\eta_1 - \eta_0)$, and $(1-\alpha)\alpha^2 + \alpha(1-\alpha)^2 = \alpha(1-\alpha)$, we have

$$\begin{aligned} &\left| \frac{(1-\alpha)\phi_x(\eta_0) + \alpha\phi_x(\eta_1) - \phi_x(\mu)}{\phi_x(\mu)} \right| \\ &= \left| \frac{\phi_x'(\mu)}{\phi_x(\mu)}[(1-\alpha)(\eta_0 - \mu) + \alpha(\eta_1 - \mu)] + \frac{1}{2}\frac{\phi_x''(\mu)}{\phi_x(\mu)}[(1-\alpha)(\eta_0-\mu)^2 + \alpha(\eta_1 - \mu)^2] + \dots \right| \\ &= \left| \frac{1}{2}\frac{\phi_x''(\mu)}{\phi_x(\mu)}\alpha(1-\alpha)(\eta_1 - \eta_0)^2 + \dots \right| \\ &\leq \sup_{z \in [\eta_0, \eta_1]} \frac{|\phi_x''(z)|}{\phi_x(\mu)} \frac{1}{2}\alpha(1-\alpha)(\eta_1 - \eta_0)^2. \end{aligned}$$

Thus,

$$\begin{aligned} \chi^2((1-\alpha)\phi_x(\eta_0) + \alpha\phi_x(\eta_1) | \phi_x(\mu)) &= \int \phi_x(\mu) \left(\frac{(1-\alpha)\phi_x(\eta_0) + \alpha\phi_x(\eta_1) - \phi_x(\mu)}{\phi_x(\mu)} \right)^2 dx \\ &\leq \left(\frac{1}{2}\alpha(1-\alpha)(\eta_1 - \eta_0)^2 \right)^2 \int \sup_{z \in [\eta_0, \eta_1]} \frac{|\phi_x''(z)|^2}{\phi_x(\mu)^2} \, \phi_x(\mu) dx. \end{aligned}$$

By distributing the square and noting that $b''(\eta) \geq 0$, we have

$$\begin{aligned}
&\int \sup_{z\in[\eta_0,\eta_1]} \frac{|\phi''_x(z)|^2}{\phi_x(\mu)^2}\ \phi_x(\mu)dx = \int \sup_{z\in[\eta_0,\eta_1]} \left(\frac{\phi_x(z)}{\phi_x(\mu)}\right)^2 (-b''(z) + (x-b'(z))^2)^2\ \phi_x(\mu)dx \\
&\quad \leq \int \sup_{z\in[\eta_0,\eta_1]} \left(\frac{\phi_x(z)}{\phi_x(\mu)}\right)^2 b''(z)^2\ \phi_x(\mu)dx + \int \sup_{z\in[\eta_0,\eta_1]} \left(\frac{\phi_x(z)}{\phi_x(\mu)}\right)^2 (x-b'(z))^4\ \phi_x(\mu)dx \\
&\quad \leq \sup_{y\in[\eta_0,\eta_1]} b''(y)^2 \int \sup_{z\in[\eta_0,\eta_1]} \left(\frac{\phi_x(z)}{\phi_x(\mu)}\right)^2\ \phi_x(\mu)dx + \int \sup_{z\in[\eta_0,\eta_1]} \left(\frac{\phi_x(z)}{\phi_x(\mu)}\right)^2 (x-b'(z))^4\ \phi_x(\mu)dx.
\end{aligned}$$

The remainder of the proof bounds the integrals. Define $\eta_- = 2\eta_0 - \mu = \eta(\theta_-)$ and $\eta_+ = 2\eta_1 - \mu = \eta(\theta_+)$. Observe that

$$\begin{aligned}
&\sup_{z\in[\eta_0,\eta_1]} \left(\frac{\phi_x(z)}{\phi_x(\mu)}\right)^2\ \phi_x(\mu) \\
&= \sup_{z\in[\eta_0,\eta_1]} h(x)\exp\big((2z-\mu)x - (2b(z)-b(\mu))\big) \\
&= \sup_{z\in[\eta_0,\eta_1]} h(x)\exp\big((2z-\mu)x - b(2z-\mu)\big)\exp\big(b(2z-\mu) - (2b(z)-b(\mu))\big) \\
&\leq e^{\kappa} \sup_{z\in[\eta_0,\eta_1]} h(x)\exp\big((2z-\mu)x - b(2z-\mu)\big) \\
&= e^{\kappa} \sup_{z\in[2\eta_0-\mu,2\eta_1-\mu]} h(x)\exp\big(zx - b(z)\big) \\
&= e^{\kappa} \sup_{z\in[\eta_-,\eta_+]} h(x)\exp\big(zx - b(z)\big) \\
&\leq e^{\kappa} \Big(\phi_x(\eta_-) + \phi_x(\eta_+) + \phi_x(\dot{b}^{-1}(x))\mathbf{1}_{x\in[\dot{b}(\eta_-),\dot{b}(\eta_+)]}\Big) \\
&\leq e^{\kappa} \Big(\phi_x(\eta_-) + \phi_x(\eta_+) + \gamma\mathbf{1}_{x\in[\dot{b}(\eta_-),\dot{b}(\eta_+)]}\Big)
\end{aligned}$$

where the second inequality follows by observing that the maximum of the function $\phi_x(z)$ will occur either at an endpoint of the interval $z \in [\eta(\theta_-), \eta(\theta_+)]$ or at the point where $\frac{\partial}{\partial z} g(z) = 0$ (if that point occurs inside the interval), and loosely bounding the maximum by simply adding the function values at all three points.

Consequently,

$$\sup_{y\in[\eta_0,\eta_1]} b''(y)^2 \int \sup_{z\in[\eta_0,\eta_1]} \left(\frac{\phi_x(z)}{\phi_x(\mu)}\right)^2\ \phi_x(\mu)dx \ \leq \sup_{\theta\in[\theta_0,\theta_1]} M_2(\theta)^2 e^{\kappa}\Big(2 + \gamma(\dot{b}(\eta_+) - \dot{b}(\eta_-))\Big).$$

By Jensen's inequality, $(a+b)^4 = 16(\frac{1}{2}a + \frac{1}{2}b)^4 \leq 8(a^4+b^4)$, so

$$\begin{aligned}\int \sup_{z\in[\eta_0,\eta_1]} \phi_x(\eta_-)(x-\dot{b}(z))^4 dx &= \int \sup_{z\in[\eta_0,\eta_1]} \phi_x(\eta_-)(x-\dot{b}(\eta_-)+\dot{b}(\eta_-)-\dot{b}(z))^4 dx \\ &\leq \int 8\phi_x(\eta_-)[(x-\dot{b}(\eta_-))^4 + \sup_{z\in[\eta_0,\eta_1]}(\dot{b}(\eta_-)-\dot{b}(z))^4]dx \\ &\leq \int 8\phi_x(\eta_-)[(x-\dot{b}(\eta_-))^4 + (\dot{b}(\eta_-)-\dot{b}(\eta_1))^4]dx \\ &= 8[M_4(\theta_-) - (\dot{b}(\eta_-)-\dot{b}(\eta_1))^4].\end{aligned}$$

Repeating an analogous series of steps for η_+, we have

$$\begin{aligned}&\int \sup_{z\in[\eta_0,\eta_1]} \left(\frac{\phi_x(z)}{\phi_x(\mu)}\right)^2 (x-b'(z))^4 \ \phi_x(\mu)dx \\ &\leq e^\kappa \int \left(\phi_x(\eta_-)+\phi_x(\eta_+)+\gamma\mathbf{1}_{x\in[\dot{b}(\eta_-),\dot{b}(\eta_+)]}\right) \sup_{z\in[\eta_0,\eta_1]}(x-\dot{b}(z))^4 dx \\ &\leq e^\kappa \left(8M_4(\theta_-) + 8(\dot{b}(\eta_1)-\dot{b}(\eta_-))^4 + 8M_4(\theta_+) + 8(\dot{b}(\eta_+)-\dot{b}(\eta_0))^4 + \tfrac{2}{5}\gamma(\dot{b}(\eta_+)-\dot{b}(\eta_-))^5\right) \\ &\leq e^\kappa \left(8M_4(\theta_-) + 8M_4(\theta_+) + 16(\dot{b}(\eta_+)-\dot{b}(\eta_-))^4 + \tfrac{2}{5}\gamma(\dot{b}(\eta_+)-\dot{b}(\eta_-))^5\right).\end{aligned}$$

The final result holds by Theorem 2. ■

A.4 Proof of Corollary 2

Proof A binomial distribution for fixed m is an exponential family $f_\theta(x) = h(x)\exp(\eta(\theta)x - b(\eta(\theta)))$ with $h(x) = \binom{m}{x}$, $\eta(\theta) = \log(\frac{\theta}{1-\theta})$, and $b(\tau) = m\log(1+e^\tau)$. Note that η is monotonically increasing, b is m-Lipschitz, and $\dot{b}(\tau) = m(1+e^{-\tau})^{-1}$ so that $\dot{b}(\eta(\theta)) = m\theta$.

Step 1: Relating θ_+, θ_- to θ_1, θ_0. I will make repeated use of the fact that if f is convex then $f(y) \geq f(x) + f'(x)^T(y-x)$. Since $\frac{x}{1-x}$ and $\frac{1-x}{x}$ are both convex, we have

$$\frac{y}{1-y} \geq \frac{x}{1-x} + \frac{y-x}{(1-x)^2} \quad \text{and} \quad \frac{1-y}{y} \geq \frac{1-x}{x} - \frac{y-x}{x^2}$$

for all $x, y \in [0,1]$.

To begin, note $\eta^{-1}(\nu) = (1+e^{-\nu})^{-1}$ so that for any θ we have $\theta(1-\theta) = \eta^{-1}(\eta(\theta))(1-\eta^{-1}(\eta(\theta))) = \frac{e^{-\eta(\theta)}}{(1+e^{-\eta(\theta)})^2}$. Observe that

$$\frac{1}{4}e^{-|\eta(\theta)|} \leq \frac{e^{-\eta(\theta)}}{(1+e^{-\eta(\theta)})^2} \leq e^{-|\eta(\theta)|}$$

and recalling that $\theta_* = \eta^{-1}((1-\alpha)\theta_0 + \alpha\theta_1) \in [\theta_0, \theta_1]$ we have

$$\begin{aligned}
\theta_+(1-\theta_+) &\geq \frac{1}{4}e^{-|\eta(\theta_+)|} = \frac{1}{4}e^{-|2\eta(\theta_1)-\eta(\theta_*)|} \\
&= \tfrac{1}{4}\mathbf{1}_{\theta_+\leq 1/2}\left(\frac{\theta_1}{1-\theta_1}\right)^2\left(\frac{1-\theta_*}{\theta_*}\right) + \tfrac{1}{4}\mathbf{1}_{\theta_+>1/2}\left(\frac{1-\theta_1}{\theta_1}\right)^2\left(\frac{\theta_*}{1-\theta_*}\right) \\
&\geq \tfrac{1}{4}\mathbf{1}_{\theta_+\leq 1/2}\left(\frac{\theta_1}{1-\theta_1}\right)^2\left(\frac{1-\theta_1}{\theta_1}\right) + \tfrac{1}{4}\mathbf{1}_{\theta_+>1/2}\left(\frac{1-\theta_1}{\theta_1}\right)^2\left(\frac{\theta_0}{1-\theta_0}\right) \\
&\geq \tfrac{1}{4}\mathbf{1}_{\theta_+\leq 1/2}\left(\frac{\theta_1}{1-\theta_1}\right) + \tfrac{1}{4}\mathbf{1}_{\theta_+>1/2}\left(\frac{1-\theta_1}{\theta_1}\right)^2\left(\frac{\theta_1}{1-\theta_1} - \frac{\theta_1-\theta_0}{(1-\theta_1)^2}\right) \\
&\geq \tfrac{1}{4}\mathbf{1}_{\theta_+\leq 1/2}\left(\frac{\theta_1}{1-\theta_1}\right) + \tfrac{1}{8}\mathbf{1}_{\theta_+>1/2}\left(\frac{1-\theta_1}{\theta_1}\right) \geq \frac{1}{8}\theta_1(1-\theta_1)
\end{aligned}$$

where the last line follows from the assumption that $\theta_1(1-\theta_1) \geq 2(\theta_1-\theta_0)$. Analogously,

$$\begin{aligned}
\theta_-(1-\theta_-) &\geq \frac{1}{4}e^{-|\eta(\theta_-)|} = \frac{1}{4}e^{-|2\eta(\theta_0)-\eta(\theta_*)|} \\
&= \tfrac{1}{4}\mathbf{1}_{\theta_-\leq 1/2}\left(\frac{\theta_0}{1-\theta_0}\right)^2\left(\frac{1-\theta_*}{\theta_*}\right) + \tfrac{1}{4}\mathbf{1}_{\theta_->1/2}\left(\frac{1-\theta_0}{\theta_0}\right)^2\left(\frac{\theta_*}{1-\theta_*}\right) \\
&\geq \tfrac{1}{4}\mathbf{1}_{\theta_-\leq 1/2}\left(\frac{\theta_0}{1-\theta_0}\right)^2\left(\frac{1-\theta_1}{\theta_1}\right) + \tfrac{1}{4}\mathbf{1}_{\theta_->1/2}\left(\frac{1-\theta_0}{\theta_0}\right)^2\left(\frac{\theta_0}{1-\theta_0}\right) \\
&\geq \tfrac{1}{4}\mathbf{1}_{\theta_-\leq 1/2}\left(\frac{\theta_0}{1-\theta_0}\right)^2\left(\frac{1-\theta_0}{\theta_0} - \frac{\theta_1-\theta_0}{\theta_0^2}\right) + \tfrac{1}{4}\mathbf{1}_{\theta_->1/2}\left(\frac{1-\theta_0}{\theta_0}\right) \\
&\geq \tfrac{1}{8}\mathbf{1}_{\theta_-\leq 1/2}\left(\frac{\theta_0}{1-\theta_0}\right) + \tfrac{1}{4}\mathbf{1}_{\theta_->1/2}\left(\frac{1-\theta_0}{\theta_0}\right) \geq \frac{1}{8}\theta_0(1-\theta_0)
\end{aligned}$$

where the last line follows from the assumption that $\theta_0(1-\theta_0) \geq 2(\theta_1-\theta_0)$. We conclude that

$$\inf_{\theta\in[\theta_-,\theta_+]}\theta(1-\theta) \geq \frac{1}{8}\inf_{\theta\in[\theta_0,\theta_1]}\theta(1-\theta). \tag{A.1}$$

Conversely,

$$\sup_{\theta\in[\theta_-,\theta_+]}\theta(1-\theta) \leq \mathbf{1}_{1/2\in[\theta_-,\theta_+]}\frac{1}{4} + \theta_+(1-\theta_+)\mathbf{1}_{\theta_+\leq 1/2} + \theta_-(1-\theta_-)\mathbf{1}_{\theta_->1/2}.$$

We consider these three cases in turn. If $\theta_+ \leq 1/2$:

$$\begin{aligned}
\theta_+(1-\theta_+) &\leq e^{-|\eta(\theta_+)|} = e^{-|2\eta(\theta_1)-\eta(\theta_*)|} \\
&= \left(\frac{\theta_1}{1-\theta_1}\right)^2\left(\frac{1-\theta_*}{\theta_*}\right) \leq \left(\frac{\theta_1}{1-\theta_1}\right)^2\left(\frac{1-\theta_0}{\theta_0}\right) \leq \left(\frac{\theta_1}{1-\theta_1}\right)^2\left(\frac{1-\theta_1}{\theta_1}+\frac{\theta_1-\theta_0}{\theta_0^2}\right) \\
&= \left(\frac{\theta_1}{1-\theta_1}\right)\left(1+\frac{\theta_1(\theta_1-\theta_0)}{(1-\theta_1)\theta_0^2}\right) \leq \left(\frac{\theta_1}{1-\theta_1}\right)\left(1+\frac{\theta_1(1-\theta_0)}{2(1-\theta_1)\theta_0}\right) \\
&= \left(\frac{\theta_1}{1-\theta_1}\right)\left(1+\frac{\theta_0(1-\theta_0)+(\theta_1-\theta_0)(1-\theta_0)}{2(1-\theta_1)\theta_0}\right) \\
&\leq \left(\frac{\theta_1}{1-\theta_1}\right)\left(1+\frac{\theta_0(1-\theta_0)+\theta_0(1-\theta_0)^2/2}{2(1-\theta_1)\theta_0}\right) \leq \frac{5}{2}\left(\frac{\theta_1}{1-\theta_1}\right) \leq 10\theta_1(1-\theta_1)
\end{aligned}$$

using the convexity of $\frac{1-x}{x}$, the assumption that $2(\theta_1-\theta_0) \leq \theta_0(1-\theta_0)$, that $\theta_1 \leq \theta_+ \leq 1/2$, and that $1-\theta_0 \leq 1$. If $\theta_- > 1/2$:

$$\begin{aligned}
\theta_-(1-\theta_-) &\leq e^{-|\eta(\theta_-)|} = e^{-|2\eta(\theta_0)-\eta(\theta_*)|} \\
&= \left(\frac{1-\theta_0}{\theta_0}\right)^2\left(\frac{\theta_*}{1-\theta_*}\right) \leq \left(\frac{1-\theta_0}{\theta_0}\right)^2\left(\frac{\theta_1}{1-\theta_1}\right) \leq \left(\frac{1-\theta_0}{\theta_0}\right)^2\left(\frac{\theta_0}{1-\theta_0}+\frac{\theta_1-\theta_0}{(1-\theta_1)^2}\right) \\
&\leq \left(\frac{1-\theta_0}{\theta_0}\right)\left(1+\frac{(1-\theta_0)(\theta_1-\theta_0)}{\theta_0(1-\theta_1)^2}\right) \leq \left(\frac{1-\theta_0}{\theta_0}\right)\left(1+\frac{(1-\theta_0)\theta_1/2}{\theta_0(1-\theta_1)}\right) \\
&= \left(\frac{1-\theta_0}{\theta_0}\right)\left(1+\frac{(1-\theta_1)\theta_1+(\theta_1-\theta_0)\theta_1}{2\theta_0(1-\theta_1)}\right) \\
&\leq \left(\frac{1-\theta_0}{\theta_0}\right)\left(1+\frac{(1-\theta_1)\theta_1+(1-\theta_1)\theta_1^2/2}{2\theta_0(1-\theta_1)}\right) \leq \frac{5}{2}\left(\frac{1-\theta_0}{\theta_0}\right) \leq 10\theta_0(1-\theta_0)
\end{aligned}$$

using the same methods as above. From these two cases, we can conclude that if $1/2 \notin [\theta_-, \theta_+]$,

$$\sup_{\theta\in[\theta_-,\theta_+]} \theta(1-\theta) \leq 10 \sup_{\theta\in[\theta_0,\theta_1]} \theta(1-\theta). \tag{A.2}$$

The remaining case, when $1/2 \in [\theta_-, \theta_+]$, also satisfies (A.2), which I now demonstrate. When $\theta_+ = 1/2$ we have $1/4 = \theta_+(1-\theta_+) \leq 10\theta_1(1-\theta_1)$ so that $\theta_1(1-\theta_1) \geq 1/40$. Because θ_1 is monotonically increasing in θ_+ and $\sup_{\theta\in[\theta_-,\theta_+]} \theta(1-\theta) \leq 1/4$ we conclude that (A.2) holds whenever $\theta_1 \leq 1/2$. A similar argument follows for all $\theta_0 \geq 1/2$. Finally, if $1/2 \in [\theta_0, \theta_1]$, it must be true that $\sup_{\theta\in[\theta_-,\theta_+]} \theta(1-\theta) \leq \sup_{\theta\in[\theta_0,\theta_1]} \theta(1-\theta)$ because $\theta_- \leq \theta_0 \leq \frac{1}{2} \leq \theta_1 \leq \theta_+$ and the function $\theta(1-\theta)$ is concave taking its maximum at $\frac{1}{2}$. Thus, (A.2) holds for all θ_-, θ_+.

We now turn our attention to bounding $\theta_+ - \theta_-$. Let $g(y) = \eta^{-1}(y)$ then $g(y) = (1+e^{-y})^{-1}$ and $\dot{g}(y) = e^{-y}(1+e^{-y})^{-2}$. Observing that $\dot{g}(\eta(\theta)) = \theta(1-\theta)$ we have by Taylor's remainder

theorem

$$\begin{aligned}\theta_+ - \theta_- &= \eta^{-1}(\eta(\theta_+)) - \eta^{-1}(\eta(\theta_-)) \leq (\eta(\theta_+) - \eta(\theta_-)) \sup_{y\in[\eta(\theta_-),\eta(\theta_+)]} e^{-y}(1+e^{-y})^{-2} \\ &= (\eta(\theta_+) - \eta(\theta_-)) \sup_{\theta\in[\theta_-,\theta_+]} \theta(1-\theta) = 2\,(\eta(\theta_1) - \eta(\theta_0)) \sup_{\theta\in[\theta_-,\theta_+]} \theta(1-\theta) \\ &\leq 20\,(\eta(\theta_1) - \eta(\theta_0)) \sup_{\theta\in[\theta_0,\theta_1]} \theta(1-\theta).\end{aligned}$$

Since $\eta(\theta) = \log(\frac{\theta}{1-\theta})$ and $\eta'(\theta) = \frac{1}{\theta} + \frac{1}{1-\theta} = \frac{1}{\theta(1-\theta)}$, we have

$$\theta_+ - \theta_- \leq 20\,(\eta(\theta_1) - \eta(\theta_0)) \sup_{\theta\in[\theta_0,\theta_1]} \theta(1-\theta) \leq 20\,(\theta_1 - \theta_0)\,\frac{\sup_{\theta\in[\theta_0,\theta_1]}\theta(1-\theta)}{\inf_{\theta\in[\theta_0,\theta_1]}\theta(1-\theta)}.$$

If $\theta_1(1-\theta_1) \geq \theta_0(1-\theta_0)$:

$$\begin{aligned}\frac{\theta_1(1-\theta_1)}{\theta_0(1-\theta_0)} &= \frac{\theta_0(1-\theta_1) + (\theta_1-\theta_0)(1-\theta_1)}{\theta_0(1-\theta_0)} \\ &\leq \frac{\theta_0(1-\theta_1) + \theta_0(1-\theta_0)(1-\theta_1)/2}{\theta_0(1-\theta_0)} \leq 1 + (1-\theta_1)/2 \leq 3/2,\end{aligned}$$

else if $\theta_0(1-\theta_0) \geq \theta_1(1-\theta_1)$

$$\begin{aligned}\frac{\theta_0(1-\theta_0)}{\theta_1(1-\theta_1)} &= \frac{\theta_0(1-\theta_1) + \theta_0(\theta_1-\theta_0)}{\theta_1(1-\theta_1)} \\ &\leq \frac{\theta_0(1-\theta_1) + \theta_0\theta_1(1-\theta_1)/2}{\theta_1(1-\theta_1)} \leq 1 + \theta_0/2 \leq 3/2.\end{aligned}$$

Finally, if $1/2 \in [\theta_0, \theta_1]$ then $\sup_{\theta\in[\theta_0,\theta_1]}\theta(1-\theta) = 1/4$ taking its maximum at $1/4$. To maximize the ratio of the sup to the inf, it suffices to just consider the case when $\theta_0 = 1/2$ or $\theta_1 = 1/2$. Thus, the above two bounds suffice for this case and we observe that

$$\frac{\sup_{\theta\in[\theta_0,\theta_1]}\theta(1-\theta)}{\inf_{\theta\in[\theta_0,\theta_1]}\theta(1-\theta)} \leq 3/2. \tag{A.3}$$

Thus, putting the pieces together, we conclude that

$$\theta_+ - \theta_- \leq 30(\theta_1 - \theta_0). \tag{A.4}$$

Step 2: Bounding γ, κ, c. In what follows, define $\theta_h = \arg\sup_{\theta\in[\theta_0,\theta_1]} \theta(1-\theta)$ and $\theta_l = \arg\inf_{\theta\in[\theta_0,\theta_1]} \theta(1-\theta)$. We now continue to bound the terms of the theorem. Note

$$\begin{aligned}
\sup_{x\in[\dot{b}(\eta(\theta_-)),\dot{b}(\eta(\theta_+))]} \phi_x(\dot{b}^{-1}(x)) &= \sup_{x\in[m\theta_-,m\theta_+]} \phi_x(\eta(x/m)) \\
&\leq \sup_{x\in[m\theta_-,m\theta_+]} \sup_{y\in[0,1]} \phi_x(\eta(y)) \\
&= \sup_{x\in[m\theta_-,m\theta_+]} \sup_{y\in[0,1]} \frac{\Gamma(m+1)}{\Gamma(m-x+1)\Gamma(x+1)} y^x(1-y)^{m-x} \\
&= \sup_{\theta\in[\theta_-,\theta_+]} \sup_{y\in[0,1]} \frac{\Gamma(m+1)}{\Gamma(m(1-\theta)+1)\Gamma(m\theta+1)} y^{m\theta}(1-y)^{m(1-\theta)} \\
&\leq \sup_{\theta\in[\theta_-,\theta_+]} \sup_{y\in[0,1]} \frac{e/2\pi}{\sqrt{m\theta(1-\theta)}} \frac{y^{m\theta}(1-y)^{m(1-\theta)}}{\theta^{m\theta}(1-\theta)^{m(1-\theta)}} \\
&= \sup_{\theta\in[\theta_-,\theta_+]} \frac{e/2\pi}{\sqrt{m\theta(1-\theta)}} \leq \frac{2}{\sqrt{m\theta_l(1-\theta_l)}} =: \gamma
\end{aligned}$$

by Stirling's approximation: $\sqrt{2\pi} \leq \frac{\Gamma(s+1)}{e^{-s}s^{s+1/2}} \leq e$ [129] and (A.1). And for any $y \in [\theta_0, \theta_1]$

$$\begin{aligned}
& b(2\eta(y) - \eta(\theta_*)) - (2b(\eta(y)) - b(\eta(\theta_*))) \\
&\quad = m\log(1 + e^{2\eta(y)-\eta(\theta_*)}) - 2m\log(1+e^{\eta(y)}) + m\log(1+e^{\eta(\theta_*)}) \\
&\quad = m\log\left(\frac{(1+e^{2\eta(y)-\eta(\theta_*)})(1+e^{\eta(\theta_*)})}{(1+e^{\eta(y)})^2}\right) \\
&\quad = m\log\left(\left(1 + \left(\frac{y}{1-y}\right)^2 \frac{1-\theta_*}{\theta_*}\right)\left(\frac{1}{1-\theta_*}\right)(1-y)^2\right) \\
&\quad = m\log\left((1-y)^2\frac{1}{1-\theta_*} + y^2\frac{1}{\theta_*}\right) \\
&\quad = m\log\left((1-2y+y^2)\frac{\theta_*}{\theta_*(1-\theta_*)} + y^2\frac{1-\theta_*}{\theta_*(1-\theta_*)}\right) \\
&\quad = m\log\left((1-2y)\frac{\theta_*}{\theta_*(1-\theta_*)} + y^2\frac{1}{\theta_*(1-\theta_*)}\right) \\
&\quad = m\log\left(1 + \frac{(y-\theta_*)^2}{\theta_*(1-\theta_*)}\right)
\end{aligned}$$

so

$$\begin{aligned}
&\sup_{y\in[\theta_0,\theta_1]} b(2\eta(y) - \eta(\theta_*)) - (2b(\eta(y)) - b(\eta(\theta_*))) \\
&\qquad \leq \sup_{y\in[\theta_0,\theta_1]} m\log\left(1 + \frac{(y-\theta_*)^2}{\theta_*(1-\theta_*)}\right) \leq m\left(\frac{(\theta_1-\theta_0)^2}{\theta_*(1-\theta_*)}\right) =: \kappa.
\end{aligned}$$

Noting that $M_2(\theta) = m\theta(1-\theta)$,

$$\begin{aligned}\sup_{y\in[\theta_0,\theta_1]} M_2(y)^2(2+\gamma(\dot{b}(\eta(\theta_+)) - \dot{b}(\eta(\theta_-)))) &\le m^2\left(\theta_h(1-\theta_h)\right)^2(2+\gamma m(\theta_+-\theta_-)) \\ &\le m^2\left(\theta_h(1-\theta_h)\right)^2\left(2+\frac{2m}{\sqrt{m\theta_l(1-\theta_l)}}30(\theta_1-\theta_0)\right) \\ &\le m^2\left(\theta_h(1-\theta_h)\right)^2\left(2+60\sqrt{m\frac{(\theta_1-\theta_0)^2}{\theta_l(1-\theta_l)}}\right).\end{aligned}$$

Since for any $\theta \in [0,1]$

$$M_4(\theta) = m\theta(1-\theta)\left(3\theta(1-\theta)(m-2)+1\right) < 3m^2\left(\theta(1-\theta)\right)^2 + m\theta(1-\theta),$$

we have

$$\begin{aligned}&8M_4(\theta_-)+8M_4(\theta_+)+16\left(\dot{b}(\eta(\theta_+))-\dot{b}(\eta(\theta_-))\right)^4+\tfrac{2}{5}\gamma\left(\dot{b}(\eta(\theta_+))-\dot{b}(\eta(\theta_-))\right)^5 \\ &\quad\le 24m^2\left(\theta_-(1-\theta_-)\right)^2+8m\theta_-(1-\theta_-)+24m^2\left(\theta_+(1-\theta_+)\right)^2+8m\theta_+(1-\theta_+) \\ &\qquad+16m^4(\theta_+-\theta_-)^4+\frac{4/5}{\sqrt{m\theta_l(1-\theta_l)}}m^5(\theta_+-\theta_-)^5 \\ &\quad\le 4800m^2\left(\theta_h(1-\theta_h)\right)^2+160m\theta_h(1-\theta_h) \\ &\qquad+3240000m^4(\theta_1-\theta_0)^4+19440000\sqrt{m\frac{(\theta_1-\theta_0)^2}{\theta_l(1-\theta_l)}}m^4(\theta_1-\theta_0)^4\end{aligned}$$

where we have applied (A.2) and (A.4). Finally, recall from above that

$$\eta(\theta_1)-\eta(\theta_0) \le \frac{\theta_1-\theta_0}{\theta_l(1-\theta_l)} \le \frac{3}{2}\frac{\theta_1-\theta_0}{\theta_*(1-\theta_*)}.$$

Step 3: Putting the pieces together. Noting that $\theta_l(1-\theta_l) \le \theta_*(1-\theta_*) \le \theta_h(1-\theta_h)$ and $\frac{\theta_h(1-\theta_h)}{\theta_l(1-\theta_l)} \le 3/2$ by (A.3), we can use $\theta_*(1-\theta_*)$ throughout at the cost of a constant. Putting it altogether, if $m\frac{(\theta_1-\theta_0)^2}{\theta_*(1-\theta_*)} \le 1$ then $\kappa \le 1$ and

$$\begin{aligned}c &\le c'\left(m^2\left(\theta_*(1-\theta_*)\right)^2+m\theta_*(1-\theta_*)+m^4(\theta_1-\theta_0)^4\right) \\ &\le c'\left(m^2\left(\theta_*(1-\theta_*)\right)^2+m\theta_*(1-\theta_*)\right)\end{aligned}$$

for some absolute constant c'. Thus,

$$\begin{aligned}
&c\left(\tfrac{1}{2}\alpha(1-\alpha)\left(\eta(\theta_1)-\eta(\theta_0)\right)^2\right)^2 \\
&\quad\le c'\left(m^2\left(\theta_*(1-\theta_*)\right)^2+m\theta_*(1-\theta_*)\right)\left(\frac{9}{8}\alpha(1-\alpha)\frac{(\theta_1-\theta_0)^2}{(\theta_*(1-\theta_*))^2}\right)^2 \\
&\quad\le c'\left(m^2+\frac{m}{\theta_*(1-\theta_*)}\right)\left(\frac{9}{8}\alpha(1-\alpha)\frac{(\theta_1-\theta_0)^2}{\theta_*(1-\theta_*)}\right)^2 \\
&\quad\le 2c'm\left(\min\{\frac{1}{m},\theta_*(1-\theta_*)\}\right)^{-1}\left(\frac{9}{8}\alpha(1-\alpha)\frac{(\theta_1-\theta_0)^2}{\theta_*(1-\theta_*)}\right)^2.
\end{aligned}$$

■

A.5 Proof of Theorem 4

Proof If $f_\theta = \mathcal{N}(\theta,\sigma^2)$ then $f_\theta(x) = h(x)\exp(\eta(\theta)x - b(\theta))$ where $h(x) = \frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{x^2}{2\sigma^2}}$, $\eta(\theta) = \frac{\theta}{\sigma^2}$, and $b(\eta(\theta)) = \frac{\eta(\theta)^2\sigma^2}{2} = \frac{\theta^2}{2\sigma^2}$. Thus,

$$\theta_* = \eta^{-1}\big((1-\alpha)\eta(\theta_0)+\alpha\eta(\theta_1)\big) = (1-\alpha)\theta_0+\alpha\theta_1$$

and

$$\sup_{y\in[\theta_0,\theta_1]} b(2\eta(y)-\eta(\theta_*)) - (2b(\eta(y)) - b(\eta(\theta_*))) = \sup_{y\in[\theta_0,\theta_1]}\frac{(y-\theta_*)^2}{\sigma^2} \le \frac{(\theta_1-\theta_0)^2}{\sigma^2} =: \kappa$$

and

$$\sup_{x\in[\dot b(\eta(\theta_-)),\dot b(\eta(\theta_+))]} f_{\dot b^{-1}(x)}(x) = \sup_{x\in[\theta_-,\theta_+]}\sup_{\theta\in\mathbb{R}}\frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{(x-\theta)^2}{2\sigma^2}} \le \frac{1}{\sqrt{2\pi\sigma^2}} =: \gamma.$$

Note that for any $\theta < \theta'$ we have $\dot b(\eta(\theta')) - \dot b(\eta(\theta)) = \theta' - \theta$, $M_2(\theta) = \sigma^2$, and $M_4(\theta) = 3\sigma^4$. Plugging these values into the theorem we have

$$\begin{aligned}
c = e^{\kappa}\Bigg(&\sup_{\theta\in[\theta_0,\theta_1]} M_2(\theta)^2\ \left(2+\gamma\left(\dot b(\eta(\theta_+)) - \dot b(\eta(\theta_-))\right)\right) \\
&+8M_4(\theta_-)+8M_4(\theta_+)+16\left(\dot b(\eta(\theta_+)) - \dot b(\eta(\theta_-))\right)^4+\tfrac{2}{5}\gamma\left(\dot b(\eta(\theta_+)) - \dot b(\eta(\theta_-))\right)^5\Bigg) \\
=&e^{\frac{(\theta_1-\theta_0)^2}{\sigma^2}}\left(\sigma^4\ \left(2+\frac{2(\theta_1-\theta_0)}{\sqrt{2\pi}\sigma}\right)+48\sigma^4+256\left(\theta_1-\theta_0\right)^4+\tfrac{64}{5\sqrt{2\pi}}\ \frac{\left(\theta_1-\theta_0\right)^5}{\sigma}\right)
\end{aligned}$$

noting that $\theta_+ - \theta_- = 2(\theta_1-\theta_0)$. If $\frac{\theta_1-\theta_0}{\sigma} \le 1$ then $c = c'\sigma^4$ for some absolute constant c' and $(\eta(\theta_1)-\eta(\theta_0))^2 = \frac{(\theta_1-\theta_0)^2}{\sigma^4}$ which yields the final result. ■

A.6 Proof of Theorem 5

First, I prove several technical lemmas necessary to analyze Algorithm 6.2.

Lemma 1 *For $i \in \mathbb{N}$, let $X_i \in [a_i, b_i]$ for $|b_i - a_i| \leq 1$ be a random variable with $\mathbb{E}[X_i] = 0$. Then*

$$\mathbb{P}\left(\bigcup_{n=1}^{\infty}\left\{\sum_{i=1}^{n} X_i \geq \alpha n + \beta\right\}\right) \leq 7\exp(-\alpha\beta/2)$$

whenever $\alpha\beta \geq 1$.

Proof First, I break the bound into two pieces:

$$\mathbb{P}\left(\bigcup_{n=1}^{\infty}\left\{\sum_{i=1}^{n} X_i \geq \alpha n + \beta\right\}\right) \leq \min_{n_0} \mathbb{P}\left(\bigcup_{n=1}^{n_0}\left\{\sum_{i=1}^{n} X_i \geq \beta\right\}\right) + \mathbb{P}\left(\bigcup_{n=n_0+1}^{\infty}\left\{\sum_{i=1}^{n} X_i \geq \alpha n\right\}\right)$$

where $\mathbb{P}\left(\bigcup_{n=1}^{n_0}\left\{\sum_{i=1}^{n} X_i \geq \beta\right\}\right) \leq \exp(-2\beta^2/n_0)$ by Doob-Hoeffding's maximal inequality. For any fixed $k \in \mathbb{N}$:

$$\mathbb{P}\left(\sum_{i=1}^{2^k} X_i \geq \alpha 2^k/2\right) \leq \exp(-\alpha^2 2^k/2)$$

and

$$\mathbb{P}\left(\bigcup_{n=2^k+1}^{2^{k+1}}\left\{\sum_{i=2^k+1}^{n} X_i \geq \alpha n/2\right\}\right) \leq \mathbb{P}\left(\bigcup_{n=2^k+1}^{2^{k+1}}\left\{\sum_{i=2^k+1}^{n} X_i \geq \alpha 2^k/2\right\}\right)$$
$$= \mathbb{P}\left(\bigcup_{\ell=1}^{2^k}\left\{\sum_{i=1}^{\ell} X_i \geq \alpha 2^k/2\right\}\right) \leq \exp(-\alpha^2 2^k/2)$$

by Hoeffding's and Doob-Hoeffding's maximal inequality, respectively. Thus

$$\begin{aligned}
&\mathbb{P}\left(\bigcup_{n=n_0}^{\infty}\left\{\sum_{i=1}^{n} X_i \geq \alpha n\right\}\right) \\
&= \mathbb{P}\left(\bigcup_{n=n_0}^{\infty}\left\{\sum_{i=1}^{2^{\lceil \log_2(n)\rceil}} X_i + \sum_{i=2^{\lceil \log_2(n)\rceil}+1}^{n} X_i \geq \alpha n\right\}\right) \\
&= \mathbb{P}\left(\bigcup_{k=\log_2(n_0)}^{\infty}\bigcup_{n=2^k+1}^{2^{k+1}}\left\{\sum_{i=1}^{2^k} X_i + \sum_{i=2^k+1}^{n} X_i \geq \alpha n\right\}\right) \\
&\leq \sum_{k=\log_2(n_0)}^{\infty} \mathbb{P}\left(\sum_{i=1}^{2^k} X_i \geq \alpha 2^k/2\right) + \mathbb{P}\left(\bigcup_{n=2^k+1}^{2^{k+1}}\left\{\sum_{i=2^k+1}^{n} X_i \geq \alpha n/2\right\}\right) \\
&\leq \sum_{k=\log_2(n_0)}^{\infty} 2\exp(-\alpha^2 2^k/2) \leq 2\int_{\log_2(n_0)}^{\infty} \exp(-(\alpha/2)^2 2^x)dx \\
&= \frac{2}{\log(2)}\int_{n_0}^{\infty} u^{-1}\exp(-(\alpha/2)^2 u)du \leq \frac{8\exp(-(\alpha/2)^2 n_0)}{n_0\alpha^2\log(2)}.
\end{aligned}$$

Putting the pieces together we have

$$\begin{aligned}
\mathbb{P}\left(\bigcup_{n=1}^{\infty}\left\{\sum_{i=1}^{n} X_i \geq \alpha n + \beta\right\}\right) &\leq \min_{n_0} \exp(-2\beta^2/n_0) + \frac{8\exp(-(\alpha/2)^2 n_0)}{n_0\alpha^2\log(2)} \\
&\leq \exp(-\beta\alpha) + \frac{4\exp(-\beta\alpha/2)}{\beta\alpha\log(2)} \leq 7\exp(-\beta\alpha/2)
\end{aligned}$$

where the last inequality holds with $\beta\alpha \geq 1$. ■

Lemma 2 *Given* $\theta_1 - \hat{\gamma} \geq \frac{2B}{m}$*,*

$$\mathbb{P}\left(\max_{j=1,\ldots,m}\sum_{s=1}^{j}(X_{i,s}-\hat{\gamma}) > B\middle|\mu_i = \theta_1\right) \geq 1 - \exp\left(-m(\theta_1-\hat{\gamma})^2/2\right).$$

Similarly, given $\hat{\gamma} - \theta_0 \geq \frac{2|A|}{m}$*,*

$$\mathbb{P}\left(\min_{j=1,\ldots,m}\sum_{s=1}^{j}(X_{i,s}-\hat{\gamma}) < A\middle|\mu_i = \theta_0\right) \geq 1 - \exp\left(-m(\hat{\gamma}-\theta_0)^2/2\right).$$

Proof We analyze the left hand side of the lemma:

$$\begin{aligned}
&\mathbb{P}\left(\max_{j=1,\ldots,m}\sum_{s=1}^{j}(X_{i,s}-\hat{\gamma})>B\middle|\mu_i=\theta_1\right)\\
&=\mathbb{P}\left(\bigcup_{j=1}^{m}\left\{\sum_{s=1}^{j}(X_{i,s}-\hat{\gamma})>B\middle|\mu_i=\theta_1\right\}\right)\\
&\geq\mathbb{P}\left(\sum_{s=1}^{m}(X_{i,s}-\hat{\gamma})>B\middle|\mu_i=\theta_1\right)\\
&=1-\mathbb{P}\left(\frac{1}{m}\sum_{s=1}^{m}(X_{i,s}-\mu_i)\leq\frac{B}{m}-(\mu_i-\hat{\gamma})\middle|\mu_i=\theta_1\right)\\
&=1-\mathbb{P}\left(\frac{1}{m}\sum_{s=1}^{m}(\mu_i-X_{i,s})\geq(\mu_i-\hat{\gamma})-\frac{B}{m}\middle|\mu_i=\theta_1\right)\\
&\geq1-\exp\left(-2m\left[(\theta_1-\hat{\gamma})-\frac{B}{m}\right]^2\right)\\
&\geq1-\exp\left(\frac{-m(\theta_1-\hat{\gamma})^2}{2}\right)
\end{aligned}$$

Where the second to last statement holds by Hoeffding's inequality, and the last uses the bound on B/m given in the lemma. A nearly identical argument yields the second half of the lemma. ■

Lemma 3 *If* $\theta_1-\hat{\gamma}\geq\frac{2B}{m}$ *then*

$$\mathbb{P}\left(\bigcup_{i=1}^{n}\left\{\mu_i=\theta_1,\max_{j=1,\ldots,m}\sum_{s=1}^{j}(X_{i,s}-\hat{\gamma})>B,\min_{j=1,\ldots,m}\sum_{s=1}^{j}(X_{i,s}-\hat{\gamma})>A\right\}\right)$$
$$\geq1-\exp\left[-\alpha n(1-\exp\left(-B(\theta_1-\hat{\gamma})\right)-7\exp(-|A|(\hat{\gamma}-\theta_0)/2))\right].$$

Proof Consider iid events Ω_i for $i=1,\ldots,n$. Then $\mathbb{P}(\bigcup_{i=1}^{n}\Omega_i)=1-\mathbb{P}(\bigcap_{i=1}^{n}\Omega_i^c)=$

$1-\mathbb{P}(\Omega_1^c)^n = 1-(1-\mathbb{P}(\Omega_i))^n \geq 1-\exp(-n\mathbb{P}(\Omega_i))$. We follow the same line of reasoning:

$$\begin{aligned}
&\mathbb{P}\left(\bigcup_{i=1}^{n}\left\{\mu_i=\theta_1,\ \max_{j=1,\ldots,m}\sum_{s=1}^{j}(X_{i,s}-\hat{\gamma})>B,\ \min_{j=1,\ldots,m}\sum_{s=1}^{j}(X_{i,s}-\hat{\gamma})>A\right\}\right)\\
&=1-\left(1-\mathbb{P}\left(\mu_i=\theta_1,\ \max_{j=1,\ldots,m}\sum_{s=1}^{j}(X_{i,s}-\hat{\gamma})>B,\ \min_{j=1,\ldots,m}\sum_{s=1}^{j}(X_{i,s}-\hat{\gamma})>A\right)\right)^n\\
&=1-\left(1-\alpha\mathbb{P}\left(\max_{j=1,\ldots,m}\sum_{s=1}^{j}(X_{i,s}-\hat{\gamma})>B,\ \min_{j=1,\ldots,m}\sum_{s=1}^{j}(X_{i,s}-\hat{\gamma})>A\Big|\mu_i=\theta_1\right)\right)^n\\
&=1-\left(1-\alpha\left(1-\mathbb{P}\left(\max_{j=1,\ldots,m}\sum_{s=1}^{j}(X_{i,s}-\hat{\gamma})<B\Big|\mu_i=\theta_1\right)\right.\right.\\
&\qquad\left.\left.-\mathbb{P}\left(\min_{j=1,\ldots,m}\sum_{s=1}^{j}(X_{i,s}-\hat{\gamma})<A\Big|\mu_i=\theta_1\right)\right)\right)^n\\
&\geq 1-\left(1-\alpha\left(1-\exp\left(-m(\theta_1-\hat{\gamma})^2/2\right)-7\exp(-|A|(\hat{\gamma}-\theta_0)/2)\right)\right)^n\\
&\geq 1-\exp\left[-\alpha n(1-\exp\left(-m(\theta_1-\hat{\gamma})^2/2\right)-7\exp(-|A|(\hat{\gamma}-\theta_0)/2))\right]\\
&\geq 1-\exp\left[-\alpha n(1-\exp\left(-B(\theta_1-\hat{\gamma})\right)-7\exp(-|A|(\hat{\gamma}-\theta_0)/2))\right]
\end{aligned}$$

Where the third-to-last inequality applies Lemmas 1 and 2. ∎

Now, these lemmas can be used to prove Theorem 5.

Proof First, I consider the estimation of $\widehat{\theta}_0$ of Algorithm 6.2, then turn to the sample complexity of the algorithm, and then prove correctness.

Let $\xi_0 = \{\widehat{\theta}_0 - \theta_0 \geq -\frac{\epsilon_0}{4}\}$ and $\xi_1 = \{\widehat{\theta}_0 - \theta_0 \leq \frac{\epsilon_0}{4}\}$ be the events that we accurately estimate the parameter θ_0. We will show that $\mathbb{P}(\xi_0) \geq 1-\delta'$ and $\mathbb{P}(\xi_1) \geq 3/4$ where $\delta' = \min\{\delta/8, \frac{1}{m\epsilon_0^2}\}$. Let $k_1 = 5$ and $k_2 = 8\epsilon_0^{-2}\log(\frac{2k_1}{\delta'})$. First note that

$$\mathbb{P}\left(\bigcup_{i=1}^{k_1}\left\{|\widehat{\mu}_{i,k_2}-\mu_i| \geq \frac{\epsilon_0}{4}\right\}\right) \leq 2k_1\exp(-2k_2(\epsilon_0/4)^2) \leq \delta'$$

so that with probability at least $1-\delta'$ we have $\widehat{\theta}_0 = \min_{i=1,\ldots,k_1}\widehat{\mu}_{i,k_2} \geq \min_{i=1,\ldots,k_1}\mu_i - \epsilon_0/4 \geq \theta_0 - \epsilon_0/4$, and in particular, $\mathbb{P}(\xi_0) \geq 1-\delta'$. Let $\mathcal{E} = \{\bigcup_{i=1}^{k_1}\{\mu_i = \theta_0\}\}$ be the event that at least one of the distributions is light. Then

$$\mathbb{P}(\mathcal{E}) = 1-\alpha^{k_1} \geq 1-2^{-k_1} \geq 31/32,$$

so that under $\mathcal{E}\cap\ \xi_0$, we have $\widehat{\theta}_0 = \min_{i=1,\ldots,k_1}\widehat{\mu}_{i,k_2} \leq \min_{i=1,\ldots,k_1}\mu_i + \epsilon_0/4 = \theta_0 + \epsilon_0/4$ which means $\mathbb{P}(\xi_1^c) \leq \mathbb{P}(\xi_0^c \cup \mathcal{E}^c) \leq \delta/8 + 1/32 \leq 1/16$. Moreover, the total number of samples is bounded by $k_1k_2 = c\epsilon_0^{-2}\log(1/\delta') \leq c\epsilon_0^{-2}\log(\max\{\frac{1}{\delta}, \log(\frac{1}{\alpha_0\delta})\})$ which is clearly dominated by $\frac{\log(1/\delta)}{\alpha_0\epsilon_0^2}$.

We now turn our attention to the sample complexity. By Wald's identitity [128, Proposition 2.18],

$$\mathbb{E}[T] = \mathbb{E}\left[\sum_{i=1}^{N} M_i\right] = \mathbb{E}[N]\mathbb{E}[M_1] = \mathbb{E}[N]((1-\alpha)\mathbb{E}[M_1|\mu_1 = \theta_0] + \alpha\mathbb{E}[M_1|\mu_1 = \theta_1]).$$

Trivially, $\mathbb{E}[N] \leq n$ and $\mathbb{E}[M_1|\mu_1 = \theta_1] \leq m$, so we only need to bound $\mathbb{E}[M_1|\mu_1 = \theta_0]$. Clearly we have that

$$\mathbb{E}[M_1|\mu_1 = \theta_0] = \mathbb{E}[M_1|\xi_0, \mu_1 = \theta_0]\mathbb{P}(\xi_0) + \mathbb{E}[M_1|\xi_0^c, \mu_1 = \theta_0]\mathbb{P}(\xi_0^c) \leq \mathbb{E}[M_1|\xi_0, \mu_1 = \theta_0] + \delta' m$$

so

$$\begin{aligned}
\mathbb{E}[M_1|\xi_0, \mu_1 = \theta_0] &\leq \sum_{t=1}^{\infty} \mathbb{P}\left(\arg\min_j \left\{\sum_{s=1}^{j}(X_{1,s} - \hat{\gamma}) < A \Big| \xi_0, \mu_1 = \theta_0\right\} \geq t\right) \\
&= \sum_{t=1}^{\infty} 1 - \mathbb{P}\left(\min_{j=1,\dots,t-1} \sum_{s=1}^{j}(X_{1,s} - \hat{\gamma}) < A \Big| \xi_0, \mu_1 = \theta_0\right) \\
&= \sum_{t=0}^{\infty} 1 - \mathbb{P}\left(\min_{j=1,\dots,t} \sum_{s=1}^{j}(X_{1,s} - \hat{\gamma}) < A \Big| \xi_0, \mu_1 = \theta_0\right) \\
&\leq \sum_{t=0}^{\infty} 1 - \mathbf{1}_{\hat{\gamma}-\theta_0 \geq \frac{2|A|}{t}} (1 - \exp\left(-t(\hat{\gamma}-\theta_0)^2/2\right) \\
&\leq \frac{2|A|}{\hat{\gamma}-\theta_0} + 2e^{1/2}(\hat{\gamma}-\theta_0)^{-2}\exp\left(-|A|(\hat{\gamma}-\theta_0)\right) \leq \frac{3|A|}{\hat{\gamma}-\theta_0} \leq \frac{293}{\epsilon_0^2}.
\end{aligned}$$

where the second inequality follows by applying Lemma 2 and the last inequality holds by ξ_0 and the value of $|A|$ since if ξ_0 holds, $\hat{\gamma} - \theta_0 = \widehat{\theta}_0 - \theta_0 + \frac{\epsilon_0}{2} \geq \frac{\epsilon_0}{4}$. Thus

$$\mathbb{E}[M_1] \leq (1-\alpha)\left[\left(\frac{293}{\epsilon_0^2}\right) + \delta' m\right] + \alpha m \leq \delta' m + \frac{1}{\epsilon_0^2}\left(293 + 64\alpha\log\left(\tfrac{14n}{\delta}\right)\right) \leq \frac{c\alpha\log\left(\frac{1}{\alpha_0\delta}\right)}{\epsilon_0^2}$$

for some c where we use the fact that $\delta' m \leq \epsilon_0^{-2}$. So we have

$$\mathbb{E}[T] \leq n\mathbb{E}[M_1] \leq \frac{c'\alpha\log(1/\alpha_0) + c''\log\left(\frac{1}{\delta}\right)}{\alpha_0\epsilon_0^2}.$$

Now, we analyze the correctness claims. Under ξ_0, $\hat{\gamma} - \theta_0 \geq \frac{\epsilon_0}{4}$. Note that this event fails to occur with probability less than $\delta/2$, and if it is used in conjunction with some other event that fails to occur with probability $\delta/2$, we may conclude that either of these events fail with probability less than δ.

To justify Claim 1, we apply Lemma 1 to observe that the probability that we output a light distribution is no greater than

$$\begin{aligned}
&\mathbb{P}(\xi_0^c) + \mathbb{P}\left(\bigcup_{i=1}^n \left\{ \max_{j=1,\dots,m} \sum_{s=1}^j (X_{i,s} - \hat{\gamma}) > B, \mu_i = \theta_0 \right\} \Big| \xi_0 \right) \mathbb{P}(\xi_0) \\
&\leq \mathbb{P}(\xi_0^c) + n(1-\alpha)\mathbb{P}\left(\max_{j=1,\dots,m} \sum_{s=1}^j (X_{i,s} - \hat{\gamma}) > B \big| \mu_i = \theta_0, \xi_0 \right) \\
&\leq \delta/2 + 7n \exp(-B(\hat{\gamma} - \theta_0)/2) \leq \delta
\end{aligned}$$

where we have used $\hat{\gamma} - \theta_0 \geq \frac{\epsilon_0}{4}$ and plugged in the values of B and n.

To justify Claim 2, assume $\alpha_0 \leq \alpha$ and $\epsilon_0 \leq \theta_1 - \theta_0$. We apply Lemma 3 to observe that the probability that we return a heavy distribution is at least

$$\begin{aligned}
&\mathbb{P}\left(\xi_0 \cap \xi_1 \cap \bigcup_{i=1}^n \left\{ \mu_i = \theta_1, \max_{j=1,\dots,m} \sum_{s=1}^j (X_{i,s} - \hat{\gamma}) > B, \min_{j=1,\dots,m} \sum_{s=1}^j (X_{i,s} - \hat{\gamma}) > A \right\} \right) \\
&= \mathbb{P}(\xi_0 \cap \xi_1)\mathbb{P}\left(\bigcup_{i=1}^n \left\{ \mu_i = \theta_1, \max_{j=1,\dots,m} \sum_{s=1}^j (X_{i,s} - \hat{\gamma}) > B, \min_{j=1,\dots,m} \sum_{s=1}^j (X_{i,s} - \hat{\gamma}) > A \right\} \Big| \xi_0, \xi_1 \right) \\
&\geq \mathbb{P}(\xi_0 \cap \mathcal{E})(1 - \exp\left[-\alpha n(1 - \exp\left(-B(\epsilon_0/4)\right) - 7\exp(-|A|(\epsilon_0/4)/2))\right]) \\
&\geq (15/16)(1 - \exp\left[-\alpha n(1 - (\tfrac{\delta}{14n})^2 - 1/3)\right]) \geq (15/16)(8/9) \geq 4/5
\end{aligned}$$

where we have used $\mathbb{P}(\xi_0 \cap \mathcal{E}) \geq 1 - \mathbb{P}(\xi_0^c) - \mathbb{P}(\mathcal{E}^c) \geq 15/16$, $(\frac{\delta}{14n})^2 \leq 1/6$, $\alpha n \geq 2\log(9)$ and plugged in the values for A and B.

To justify Claim 3, we simply observe that the algorithm always terminates after $n \times m$ steps. ■

A.7 Proof of Theorem 6

Proof The proof is broken up into a few steps, summarized as follows. For any given α_0, ϵ_0, Theorem 5 takes just $O\left(\frac{\alpha\log(1/\alpha_0)+\log(1/\delta)}{\alpha_0\epsilon_0^2}\right)$ samples in expectation and the procedure makes an error (i.e. returns a light distribution) with probability less than δ. Define $\epsilon = \theta_1 - \theta_0$. In addition, if$\epsilon = \theta_1 - \theta_0$, $\alpha \geq \alpha_0$, and $\epsilon \geq \epsilon_0$ then with probability at least $4/5$ a heavy distribution is returned after the same expected number of samples. We will leverage this result to show that if we are given an upper bound γ_0 such that $\frac{1}{\alpha\epsilon^2} \leq \gamma_0$ then it is possible to identify a heavy distribution with probability at least $4/5$ using just $O\left(\log_2(\gamma_0)\gamma_0\left[\alpha\log_2(\gamma_0) + \log(\log_2(\gamma_0)/\delta)\right]\right)$ samples in expectation. Finally, we apply the "doubling trick" to γ so that even though the tightest γ is not known a priori, we can adapt to it using only twice the number of samples as if we had known it. Because each of the stages is independent of one another, the probability that the procedure requires more than $\ell_* + i$ stages is less than $(1/5)^i$, which yields our expected sample complexity.

For all $\ell \in \mathbb{N}$ define $\delta_\ell = \frac{\delta}{2\ell^3}$ and $\gamma_\ell = 2^\ell$. Fix some ℓ and consider the set $\{(\alpha,\epsilon) : \frac{1}{\alpha\epsilon^2} = \gamma_\ell\}$. Clearly, in this set, $\alpha \in [1/\gamma_\ell, 1/2]$. For all $k \in \{0,\ldots,\ell-1\}$, define $\alpha_k = \frac{2^k}{\gamma_\ell}$ and $\epsilon_k = \sqrt{\frac{1}{2\alpha_k\gamma_\ell}}$. The key observation is that

$$\{(\alpha,\epsilon) : \frac{1}{\alpha\epsilon^2} \leq \gamma_\ell\} \subseteq \bigcup_{k=0}^{\log_2\gamma_\ell - 1} \{(\alpha,\epsilon) : \alpha \geq \alpha_k, \epsilon \geq \epsilon_k\}. \tag{A.5}$$

To see this, fix any (α',ϵ') such that $\frac{1}{\alpha'\epsilon'^2} \leq \gamma_\ell$. Let k_* be the integer that satisfies $\alpha_{k_*} \leq \alpha' < 2\alpha_{k_*}$. Such a k_* must exist since $\alpha_{\ell-1} = \frac{1}{2} \geq \alpha' \geq \frac{1}{\gamma_\ell\epsilon'^2} \geq \frac{1}{\gamma_\ell} = \alpha_0$. Then $\gamma_\ell \geq \frac{1}{\alpha'\epsilon'^2} \geq \frac{1}{2\alpha_{k_*}\epsilon'^2}$ which means $\epsilon' \geq \sqrt{\frac{1}{2\alpha_{k_*}\gamma_\ell}} = \epsilon_{k_*}$ which proves the claim of (A.5). Consequently, even if no information about α or ϵ individually is known but $\frac{1}{\alpha\epsilon^2} \leq \gamma_\ell$, one can cover the entire range of valid (α,ϵ) with just $\log_2(\gamma_\ell) = \ell$ landmarks (α_k,ϵ_k).

For any $\ell \in \mathbb{N}$ and $k \in \{0,\ldots,\ell-1\}$, if Algorithm 6.2 is used with $\alpha_0 = \alpha_k, \epsilon_0 = \epsilon_k$ and $\delta = \delta_\ell$ then the probability that a light distribution is returned, declared heavy is less than δ_ℓ. And the probability that a light distribution is returned, declared heavy for *any* $\ell \in \mathbb{N}$ and $k \in \{0,\ldots,\ell-1\}$ is less than $\sum_{\ell=1}^\infty \ell\delta_\ell = \delta\sum_{\ell=1}^\infty \ell/(2\ell^3) \leq \delta$. Thus, given that Algorithm 6.3 terminates with a non-null distribution h, h is heavy with probability at least $1-\delta$. This proves correctness. We next bound the expected number of samples taken before the procedure terminates.

With the inputs given in the last paragraph for any k,ℓ, Algorithm 6.2 takes an expected number samples bounded by $c\gamma_\ell(\alpha\log(1/\alpha_k) + \log(1/\delta_\ell))$. Let $L \in \mathbb{N}$ be the random stage at which Algorithm 6.3 terminates with a non-null distribution h. Let ℓ_* be the first integer such that there exists a $k \in \{0,\ldots,\ell_*-1\}$ with $\alpha \geq \alpha_k$ and $\epsilon \geq \epsilon_k$ (recall that in this case $\frac{1}{\alpha_k\epsilon_k^2} \leq \gamma_{\ell_*}$). Then by the end of stage $\ell \geq \ell_*$, at most $c\ell\gamma_\ell(\alpha\log(\gamma_\ell) + \log(1/\delta_\ell))$ samples in expectation were taken on stage ℓ and with probability at least $4/5$ the procedure terminated with a heavy coin. By the independence of samples between rounds, observe that $\mathbb{P}(L \geq \ell_* + i) = \sum_{j=i}^\infty \mathbb{P}(L = \ell_* + j) \leq (\frac{5}{4})(\frac{1}{5})^i$. Thus, if M_ℓ is the number of samples taken at stage ℓ then by Wald's identify, the total expected number of samples taken before

termination is bounded by

$$\begin{aligned}
&\mathbb{E}\left[\sum_{\ell=1}^{L} c\ell\gamma_\ell(\alpha\log(\gamma_\ell)+\log(1/\delta_\ell))\right] \\
&= \sum_{\ell=1}^{\infty}\mathbb{E}[M_\ell]\mathbb{P}(L\geq\ell) \leq \sum_{\ell=1}^{\infty} c\ell\gamma_\ell(\alpha\log(\gamma_\ell)+\log(1/\delta_\ell))\mathbb{P}(L\geq\ell) \\
&\leq \sum_{\ell=1}^{\ell_*} c\ell\gamma_\ell(\alpha\log(\gamma_\ell)+\log(1/\delta_\ell)) + \sum_{\ell=\ell_*+1}^{\infty} c\ell\gamma_\ell(\alpha\log(\gamma_\ell)+\log(1/\delta_\ell))(\tfrac{5}{4})(\tfrac{1}{5})^{\ell-\ell_*} \\
&\leq \sum_{\ell=1}^{\ell_*} c\ell 2^\ell(\alpha\ell+\log(2\ell^3/\delta)) + \sum_{\ell=\ell_*+1}^{\infty} c\ell 2^\ell(\alpha\ell+\log(2\ell^3/\delta))(\tfrac{5}{4})(\tfrac{1}{5})^{\ell-\ell_*} \\
&\leq c\ell_*(\alpha\ell_*+\log(2\ell_*^3/\delta))\sum_{\ell=1}^{\ell_*}2^\ell + c(\tfrac{5}{4})5^{\ell_*}\sum_{\ell=\ell_*+1}^{\infty}\left(\alpha\ell^2(\tfrac{2}{5})^\ell + 3\ell\log(\ell)(\tfrac{2}{5})^\ell + \log(2/\delta)\ell(\tfrac{2}{5})^\ell\right) \\
&\leq 2c\ell_*2^{\ell_*}(\alpha\ell_*+\log(2\ell_*^3/\delta)) \\
&\quad + c(\tfrac{5}{4})5^{\ell_*}\left(2\alpha(\ell_*+1)^2(\tfrac{2}{5})^{\ell_*} + 12\log(2e^2\ell_*)(\ell_*+1)(\tfrac{2}{5})^{\ell_*} + 4\log(2/\delta)(\ell_*+1)(\tfrac{2}{5})^{\ell_*}\right) \\
&\leq c'\ell_*2^{\ell_*}(\alpha\ell_*+\log(\ell_*)+\log(1/\delta))
\end{aligned}$$

for some absolute constant c' since $\sum_{k=n}^{\infty} ka^k \leq \frac{na^n}{(1-a)^2}$, $\sum_{k=n}^{\infty} k^2a^k \leq \frac{n^2a^n}{(1-a)^3}$, and

$$\begin{aligned}
\sum_{\ell=\ell_*+1}^{\infty}\ell\log(\ell)(\tfrac{2}{5})^\ell &\leq \log(2\ell_*)\sum_{\ell_*+1}^{2\ell_*}\ell(\tfrac{2}{5})^\ell + \sum_{2\ell_*+1}^{\infty}\ell(\tfrac{2}{5})^{\ell/2}\left(\log(\ell)(\tfrac{2}{5})^{\ell/2}\right) \\
&\leq \log(2e^2\ell_*)\sum_{\ell_*+1}^{\infty}\ell(\tfrac{2}{5})^\ell \leq 4\log(2e^2\ell_*)(\ell_*+1)(\tfrac{2}{5})^{\ell_*}
\end{aligned}$$

since $\max_{x\geq1}\log(x)(\frac{2}{5})^{x/2}\leq 1$. Noting that $\ell_*\leq\log_2(\frac{1}{\alpha\epsilon^2})+1$, we have that the total number of samples, in expectation, is bounded by

$$\begin{aligned}
c'\ell_*2^{\ell_*}(\alpha\ell_*+\log(\ell_*)+\log(1/\delta)) &\leq c''\frac{\log_2(\frac{1}{\alpha\epsilon^2})}{\alpha\epsilon^2}(\alpha\log_2(\tfrac{1}{\alpha\epsilon^2})+\log(\log_2(\tfrac{1}{\alpha\epsilon^2}))+\log(1/\delta)) \\
&\leq c'''\frac{\log_2(\frac{1}{\alpha\epsilon^2})}{\alpha\epsilon^2}(\alpha\log_2(\tfrac{1}{\epsilon^2})+\log(\log_2(\tfrac{1}{\alpha\epsilon^2}))+\log(1/\delta))
\end{aligned}$$

where we've used the fact that $\sup_{\alpha\in[0,1]}\alpha\log(1/\alpha)\leq e^{-1}$. ■

A.8 Proof of Theorem 7

Proof Let $\widehat{\mu}_i$ be the empirical mean of the ith distribution sampled m times with mean $\mu_i\in\{\theta_0,\theta_1\}$. Let N be the minimum of $\widehat{n}$ and the first $i\in\mathbb{N}$ such that $\widehat{\mu}_i\geq\frac{\theta_0+\theta_1}{2}$. Declare distribution N to be heavy. The total number of flips this procedure makes equals mN.

Define the events

$$\xi_1 = \bigcup_{i=1}^{\widehat{n}} \{\mu_i = \theta_1\}, \quad \text{and} \quad \xi_2 = \bigcap_{i=1}^{\widehat{n}} \{|\widehat{\mu}_i - \mu_i| < \tfrac{\theta_1-\theta_0}{2}\}.$$

Note that $\mathbb{P}(\xi_1^c) = \mathbb{P}(\mu_1 = \theta_0)^{\widehat{n}} = (1-\alpha)^{\widehat{n}} \leq \exp(-\alpha\widehat{n}) \leq \delta/2$. And, by a union bound and Chernoff's inequality $\mathbb{P}(\xi_2^c) \leq 2\widehat{n}e^{-m(\theta_1-\theta_0)^2/2} \leq \delta/2$. Thus, the probability that ξ_1 or ξ_2 fail to occur is less than δ, so in what follows assume they succeed.

Under ξ_1 at least one of the $\widehat{n}$ distributions is heavy. Under ξ_2, for any $i \in [\widehat{n}]$ with $\mu_i = \theta_0$ we have $\hat{\mu}_i < \mu_i + \frac{\theta_1-\theta_0}{2} = \frac{\theta_0+\theta_1}{2}$ which implies that the procedure will never exit with a light distribution unless $N = \widehat{n}$. On the other hand, for the first $i \in [\widehat{n}]$ with $\mu_i = \theta_1$ we have $\hat{\mu}_i > \mu_i - \frac{\theta_1-\theta_0}{2} = \frac{\theta_0+\theta_1}{2}$ which means the algorithm will output distribution i at time $N = i$. Thus, N is equal to the first distribution that is heavy and

$$\begin{aligned}
\mathbb{E}[N] &= \sum_{n=1}^{\widehat{n}} \mathbb{P}(N \geq n) = \sum_{n=1}^{\widehat{n}} \mathbb{P}(N \geq n, \max_{i=1,\dots,n-1} \mu_i \neq \theta_1) + \mathbb{P}(N \geq n, \max_{i=1,\dots,n-1} \mu_i = \theta_1) \\
&\leq \sum_{n=1}^{\widehat{n}} \mathbb{P}(\max_{i=1,\dots,n-1} \mu_i \neq \theta_1) \\
&\qquad + \mathbb{P}(\cup_{i=1}^{n-1}\{|\widehat{\mu}_i - \mu_i| > \tfrac{\theta_1-\theta_0}{2}\} | \max_{i=1,\dots,n-1} \mu_i = \theta_1)\mathbb{P}(\max_{i=1,\dots,n-1} \mu_i = \theta_1) \\
&\leq \sum_{n=1}^{\widehat{n}} \mathbb{P}(\max_{i=1,\dots,n-1} \mu_i \neq \theta_1) + \mathbb{P}(\cup_{i=1}^{n-1}\{|\widehat{\mu}_i - \mu_i| > \tfrac{\theta_1-\theta_0}{2}\}) \\
&\leq \sum_{n=1}^{\widehat{n}} (1-\alpha)^{n-1} + \tfrac{n-1}{\widehat{n}}\frac{\delta}{2} \leq \frac{1}{\alpha} + \widehat{n}\delta/4 = \frac{1}{\alpha}(1 + \tfrac{\delta\log(2e/\delta)}{4}) \leq \frac{3/2}{\alpha}.
\end{aligned}$$

Multiplying $\mathbb{E}[N]$ by m yields the result. ■

A.9 Proof of Theorem 8

Proof On each stage k, Algorithm 6.2 is called with $\delta/(2k^2)$. By the guarantees of Theorem 5, the probability that Algorithm 6.4 ever outputs a light distribution is less than $\sum_{k=1}^{\infty} \delta/(2k^2) \leq \delta$. Thus, if a distribution is output, it is heavy with probability at least $1-\delta$. We now show that the expected number of samples taken before outputting a distribution is bounded.

Let K be the random stage in which Algorithm 6.4 outputs a distribution and let k_* be the smallest $k \in \mathbb{N}$ that satisfies $2^{-k} \leq \theta_1 - \theta_0$. By the guarantees of Theorem 5 and the independence of the stages k, $\mathbb{P}(K \geq k_* + i) \leq \sum_{\ell=i}^{\infty}(\frac{1}{5})^{\ell} = (\frac{5}{4})(\frac{1}{5})^i$. Moreover, if M_k is the number of measurements taken at stage k, then by Wald's identity the expected number of

measurements is bounded by

$$\mathbb{E}\left[\sum_{k=1}^{K} M_k\right] = \sum_{k=1}^{\infty}\mathbb{E}[N_k]\mathbb{P}(K \geq k) \leq \sum_{k=1}^{\infty}\frac{c'\alpha\log(1/\alpha) + c''\log\left(\frac{2k^2}{\delta}\right)}{\alpha 2^{-2k}}\max\{1, (\tfrac{5}{4})(\tfrac{1}{5})^{k-k_*}\}$$
$$\leq \sum_{k=1}^{k_*}\frac{c'''\log\left(\frac{2k_*^2}{\delta}\right)}{\alpha}4^k + 5^{k_*}\tfrac{c'''}{\alpha}\sum_{k=k_*+1}^{\infty}\left(2\log(k)+\log(\tfrac{2}{\delta})\right)(\tfrac{4}{5})^k$$
$$\leq \frac{c'''\log\left(\frac{2k_*^2}{\delta}\right)}{\alpha}4^{k_*+1} + 5^{k_*}\tfrac{c'''}{\alpha}\sum_{k=k_*+1}^{\infty}\left(2\log(k)+\log(\tfrac{2}{\delta})\right)(\tfrac{4}{5})^k \leq \frac{c''''\log\left(\frac{k_*}{\delta}\right)}{\alpha}(2^{k_*})^2$$

since $\sup_\alpha \alpha\log(1/\alpha) \leq e^{-1}$ and

$$\sum_{k=k_*}^{\infty}\log(k)(\tfrac{4}{5})^k = \sum_{k=k_*}^{2k_*-1}\log(k)(\tfrac{4}{5})^k + \sum_{k=2k_*}^{\infty}\log(k)(\tfrac{4}{5})^{k/2}(\tfrac{4}{5})^{k/2}$$
$$\leq \log(2k_*)\sum_{k=k_*}^{2k_*-1}(\tfrac{4}{5})^k + \sum_{k=2k_*}^{\infty}(\tfrac{4}{5})^{k/2} \leq (\log(2k_*)+2)\sum_{k=k_*}^{\infty}(\tfrac{4}{5})^k = 5\log(2e^2k_*)(\tfrac{4}{5})^{k_*}$$

since $\sup_k \log(k)(\frac{4}{5})^{k/2} \leq 1$. Noting that $k_* \leq \log_2(\frac{1}{\theta_1-\theta_0}) + 1$ completes the proof. ■

A.10 Proof of Theorem 9

Proof The proof of this result is nearly identical to that of Theorem 8 except the following changes. Let K be the random stage in which Algorithm 6.5 outputs a distribution and let k_* be the smallest $k \in \mathbb{N}$ that satisfies $2^{-k} \leq \alpha$. Moreover, if M_k is the number of measurements taken at stage k, then by Wald's identity expected number of measurements is bounded by

$$\mathbb{E}\left[\sum_{k=1}^{K} M_k\right] = \sum_{k=1}^{\infty}\mathbb{E}[N_k]\mathbb{P}(K \geq k) \leq \sum_{k=1}^{\infty}\frac{c'\alpha\log(2^k) + c''\log\left(\frac{2k^2}{\delta}\right)}{2^{-k}\epsilon^2}\max\{1, (\tfrac{5}{4})(\tfrac{1}{5})^{k-k_*}\}$$
$$\leq \sum_{k=1}^{k_*}\frac{c'''\log\left(\frac{2k_*^2}{\delta}\right)}{\epsilon^2}2^k + 5^{k_*}\tfrac{c'''}{\epsilon^2}\sum_{k=k_*+1}^{\infty}\left(\alpha k\log(2) + 2\log(k)+\log(\tfrac{2}{\delta})\right)(\tfrac{2}{5})^k$$
$$\leq \frac{c''''\left(\alpha k_* + \log\left(\frac{k_*}{\delta}\right)\right)}{\epsilon^2}2^{k_*} \leq \frac{c'''''\log\left(\log(1/\alpha)/\delta\right)}{\alpha\epsilon^2}$$

by the same series of steps as the proof of Theorem 8 and the fact that $\sum_{k=n}^{\infty} ka^k \leq \frac{na^n}{(1-a)^2}$ for any $a \in (0,1)$. The final inequality follows from $k_* \leq \log_2(1/\alpha) + 1$ and that $\alpha k_* = \alpha\log_2(2/\alpha) \leq 2$. ■