TECHNIQUES FOR QUALITY CONTROL IN APPLICATIONS THAT USE CROWDSOURCED INPUT

A dissertation submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Vasileios Polychronopoulos

March 2017

The Dissertation of Vasileios Polychronopoulos
is approved:

__________________________________________
Professor Luca de Alfaro, Chair

__________________________________________
Professor James Davis

__________________________________________
Dr. Madhukar Thakur

__________________________________________
Tyrus Miller
Vice Provost and Dean of Graduate Studies
# Table of Contents

List of Figures vii

List of Tables viii

Abstract ix

Acknowledgments xii

1 Introduction 1

1.1 Quality control in crowdsourcing .............................................. 1
1.2 Outline of the dissertation ...................................................... 3

2 Efficient techniques for Crowdsourced Top-k Lists 6

2.1 Problem setting and contributions ........................................... 6
2.2 Notations ................................................................................. 12
  2.2.1 Preliminaries. ................................................................. 12
  2.2.2 The top-k problem. .......................................................... 13
2.3 Top-k Tournament Algorithm .................................................. 15
  2.3.1 Algorithm Definition ......................................................... 17
  2.3.2 Aggregating Answers to s-Ranking Tasks .............................. 19
  2.3.3 Determining the Number of Tasks ...................................... 23
  2.3.4 Speculative optimization for reduced latency ...................... 26
  2.3.5 Redistributive optimization .............................................. 27
2.4 Experimental study of the tournament algorithm ..................... 28
  2.4.1 Simulations ..................................................................... 28
  2.4.2 Experiments over Mechanical Turk ................................. 36
2.5 Large top-k lists ...................................................................... 40
2.6 A Recursive Crowdsourced Top-k Algorithm ......................... 42
  2.6.1 Endgame top-k algorithms ............................................... 49
2.7 Handling inaccuracy of crowds .............................................. 51
  2.7.1 Distributing budget to ranking tasks ................................. 52
2.8 Randomized variant ............................................................... 56
## 3 Reliable Aggregation of Boolean Crowdsourced Tasks

3.1 Problem setting and contributions .................................. 71
3.2 Notation and previous work ........................................ 75
   3.2.1 Notation .................................................. 75
   3.2.2 Previous work ............................................ 75
   3.2.3 Beta belief distributions for users and items ............... 78
   3.2.4 Iterative updates of user and items distribution .......... 81
   3.2.5 A beta-distribution interpretation of KOS ................. 82
   3.2.6 Limitations of KOS approach .............................. 84
3.3 Two proposed algorithms: Regularized Harmonic and Beta Shape Parameter Estimation .................. 87
   3.3.1 Regularized Harmonic Algorithm .......................... 88
   3.3.2 Beta Shape Parameter Estimation Algorithm ............... 89
   3.3.3 Obtaining the bayesian updates ........................... 92
   3.3.4 Parameter estimation for the beta distribution .......... 93
   3.3.5 Extention to the case of multiple votes .................... 95
3.4 Experimental study on synthetic data ............................... 99
   3.4.1 Independent users model ................................. 99
   3.4.2 The limited sources model ............................... 101
   3.4.3 Algorithms ............................................. 101
   3.4.4 Results ................................................ 103
3.5 Experimental study on real-world data ............................ 106
   3.5.1 Wikipedia edits Mechanical Turk data ..................... 112
   3.5.2 NLP Mechanical Turk datasets ............................ 115
   3.5.3 The Duchenne experiment dataset .......................... 116
   3.5.4 Average recall for real-world datasets ................... 116
3.6 Conclusions .................................................. 118

## 4 Learning from Crowdsourced Graphs Using LSTMs

4.1 Problem setting and contributions ............................... 119
4.2 Related Work ............................................... 125
4.3 Learning from Graph Neighborhoods ............................ 128
   4.3.1 Multi-Level Sequence Learners ......................... 131
   4.3.2 Training .............................................. 135
4.4 Applications ............................................. 135
4.4.1 Crowdsourcing boolean labels ........................................... 136
4.4.2 Peer Grading ............................................................... 138
4.4.3 Prediction of Bitcoin Spending ........................................ 139
4.4.4 Discussion ................................................................. 141

Bibliography ................................. 143
List of Figures

2.1 Tournament algorithm for the top-k problem. ......................... 16
2.2 Pseudocode for median rank aggregation .............................. 21
2.3 Example of a swap in the baseline ranking due to distorted perceptions of workers about the ranks of items ......................... 30
2.4 Comparative performance for increasing budget ........................ 32
2.5 Comparative performance for increasing budget with a high error rate of workers ....................................................... 33
2.6 Performance of the tournament algorithm using the adaptive task allocation technique and the speculative optimization for different levels of error rate of workers ............................................. 34
2.7 Performance of the tournament algorithm using the adaptive task allocation technique and the speculative optimization for different levels of spam .......................................................... 35
2.8 Comparative performance of the tournament algorithm with the Compare operator ......................................................... 36
2.9 Comparative performance of our adaptive tournament algorithm against the tournament max algorithm .................................... 37
2.10 An example of a completed ranking task of 10 polygons, that we provided to the human workers for illustrative purposes at the preamble of the HTML form of the Human Intelligence Tasks ................................. 38
2.11 An actual ranking task of a set of 10 polygons that human workers had to complete as part of our Mechanical Turk experiments. ........... 39
2.12 Comparative performance of the basic variant of the tournament algorithm, against the variant using the adaptive task allocation technique using workers from Mechanical Turk ..................................... 40
2.13 Recursive algorithm Crowd-Top-k for the top-k problem ............ 44
2.14 Example of execution of the Crowd-Top-k algorithm to obtain the top-5 list of an itemset of 320 items using crowdsourced ranking tasks of size 4 ................................................................. 46
2.15 Comparative performance of the Crowd-Top-k algorithm and the tournament algorithm for increasing error rate ............................ 61
2.16 Comparative performance of methods for large top-k lists ............ 63
2.17 Performance of budgeting strategy ........................................ 64
2.18 Error for increasing probability of loss ........................................ 66
2.19 Example of an increased difficulty pairwise comparison task with polygons addressed to real workers of Amazon Mechanical Turk ........................................ 67
2.20 Comparative performance using Amazon’s Mechanical Turk crowd ........................................ 69

3.1 KOS algorithm for labeling items using binary crowdsourced answers ........................................ 77
3.2 Regularized Harmonic Algorithm ........................................ 90
3.3 Beta Shape Parameter estimate (BSP) algorithm ........................................ 100
3.4 Results for a 100×100 regular bipartite graph with 5 votes per item using the spammer-hammer model ........................................ 102
3.5 Results for a 100×100 regular bipartite graph with uniformly distributed user accuracies ........................................ 103
3.6 Results for a 100×100 regular bipartite graph with beta distributed user accuracies ........................................ 106
3.7 Results for a 100×100 regular bipartite graph with the limited sources model ........................................ 107
3.8 Results for a 100×100 Pareto distribution graph with the limited sources model ........................................ 108
3.9 Results for a 1000×1000 regular bipartite graph with beta distributed user accuracies ........................................ 109
3.10 Results for a 1000×1000 Pareto distribution graph with the limited sources model ........................................ 110
3.11 Comparative performance with increasing number of votes per item ........................................ 111
3.12 Comparative performance with varying skew of the two classes ........................................ 111
3.13 Comparative study for the Wikipedia edits Mechanical Turk dataset ........................................ 112
3.14 Comparative performance for the RTE dataset ........................................ 114
3.15 Comparative performance for the temp dataset ........................................ 114
3.16 Comparative performance for the Duchenne experiment dataset ........................................ 115

4.1 Example of a bipartite graph of users and assignments where users grade the work of their peers ........................................ 120
4.2 The transaction history of a Bitcoin address ........................................ 121
4.3 Example of a graph of users interacting over Wikipedia pages ........................................ 122
4.4 An example of a graph and its asymmetric tree unfolding ........................................ 130
4.5 Forward propagation corresponding to the tree unfolding of Figure 4.4 ........................................ 134
List of Tables

3.1 Comparative average recall for real-world datasets . . . . . . . . . . . . 117
4.1 Performance of KOS [34], EM (Expectation Maximization) and multi-
level sequence learners (MLSLs) of different depths. . . . . . . . . . . . 138
4.2 Performance of EM and 1,2-depth MLSL on peer grading data. . . . . 139
4.3 The prediction results on blockchain addresses using baseline approach,
and MLSL of depths 1, 2, 3. . . . . . . . . . . . . . . . . . . . . . . . . . 140
Abstract

Techniques for Quality Control in Applications that Use Crowdsourced Input

by

Vasileios Polychronopoulos

Crowdsourcing is the use of human workers, usually through the Internet, for obtaining useful services and performing computation tasks for which automated computation is inefficient or inapplicable. Crowdsourced input usually has a monetary cost and is obtained through the use of a crowdsourcing marketplace. Applications with crowdsourced input have to address several challenges: obtaining worker input has high latency; workers may disagree on the same tasks and some workers may provide wrong input on purpose. We introduce novel methods that provide state-of-the-art solutions to the quality control problem of crowdsourcing in the context of different applications. Some of the most basic and common applications of crowdsourcing concern ranking and labeling problems. In ranking problems, the input of the crowd is used to sort elements in ranked order; whereas in labeling, the input of the crowd is used to assign each element a label. We study here two of the most common instances of these problems, providing algorithms that advance the state of the art: the top-k ranking problem, and the boolean labeling problem. For crowdsourced top-k lists the goal is, for some number k, to obtain the top-k items out of larger itemsets, using human workers to perform comparisons among items. An example application of the top-k problem is to short-list a large set of college applications using advanced students as workers.
We evaluate our proposed techniques for the top-k problem against prior art using simulations as well as real crowds in Amazon Mechanical Turk. A randomized variant of the proposed algorithms achieves significant budget saves, especially for very large itemsets and large top-k lists, with negligible risk of lowering the quality of the output. The boolean labeling problem consists in aggregating boolean (yes/no) answers by the crowd reliably; such tasks are widely used, for instance, to label input as spam or to judge the appropriateness of web content. We introduce two unsupervised algorithms for this problem: one derived heuristically which is simple to implement, and one based on iterated bayesian parameter estimation of user reputation models. We provide mathematical insight into the benefits of the proposed algorithms over existing approaches, and we validate these insights by showing that both algorithms offer improved performance on several occasions on both synthetic datasets and real-world datasets obtained via Amazon Mechanical Turk. In some applications, there may be access to a set of informative features on workers, on their tasks and on how workers complete tasks (e.g., the time a worker took to complete a task). Our third contribution consists in techniques for enhancing the precision of any crowdsourcing task. At the core of each crowdsourcing task is the consideration of user input on items of work: fundamentally, this constitutes a bipartite graph between users and items, on which some inference must be performed. We introduce a machine-learning approach that works directly on the graph of workers, tasks and the associated features, utilizing a multi-level architecture of Long Short-Term Memory neural nets (LSTMs), without the need for resorting to a-priori models of user behavior or item dynamics. We show that
when such informative features are present, the machine-learning approach can provide enhanced quality compared to model-based inference approaches.
Acknowledgments

I want to thank my academic advisor Luca de Alfaro, who helped me in so many ways. I also thank the members of the committee for their support and their insightful feedback, and all the people who did research with me the past years. Last but not least, I want to thank the friends I met in Santa Cruz who supported me in difficult times and shared my joy in good times. They know who they are.
Chapter 1

Introduction

1.1 Quality control in crowdsourcing

Despite the past expectations about the advances of Artificial Intelligence, humans are better than automated computation in many tasks, for example, judging appropriateness of content of a picture. Crowdsourcing is a term introduced in 2005, to describe the use of human workers through the Internet to complete tasks in bulk. The tasks are named Human Intelligence Tasks (HITs). Crowdsourcing is now in wide use by organizations and individuals allowing them to obtain input from human agents on problems for which automated computation is not applicable or prohibitively costly. Due to the involvement of the human factor, several challenges come with the use of crowdsourcing. Poor quality feedback from users is common due to malevolence of humans or due to misunderstanding of tasks by the crowd. On the other hand, humans can honestly give different answers to the same task, due to different perceptions, without
one view being objectively more valid than another. Crowdsourced tasks have a monetary cost in most cases, the cost to pay the workers that perform the tasks. Occasionally, workers can take significantly long time to complete otherwise short tasks, increasing the latency of the application. Having a single highly belated task in a batch of several parallel tasks can cancel out the benefits of parallelism in saving time, since the latency of the batch is the latency of the task that took longer to execute. Applications that use crowdsourcing aim to optimize monetary cost, latency and quality of results. There is a three-way tradeoff between these three quantities; optimizing one comes at the cost of the other two. The precise nature of this tradeoff is application-specific and not fully understood.

While the industrial users of crowdsourcing already employ ad hoc solutions to its challenges, the past years there are efforts by computer science researchers to address the challenges of crowdsourcing in a principled way. The challenges acquire different characteristics depending on the application for which human input is used and the crowds in question.

Research into crowdsourcing has established a new field of computer science [41], importing and building on notions from traditional domains of computer science such as query optimization and distributed systems to theoretic frameworks like social choice theory [58] and game theory. In this dissertation we introduce techniques that obtain reliable crowdsourced results by optimizing available crowdsourced input by inferring reliable input and filtering out spam and low quality input.
1.2 Outline of the dissertation

In Chapter 2, we describe efficient techniques for crowdsourced top-\(k\) lists. Obtaining top-\(k\) lists is the same as short-listing a large set of items, which is a natural and expensive operator. We initially investigate tournament algorithms that are applicable to small top-\(k\) lists and devise a set of techniques for making them robust. We validate the robustness of the proposed techniques with a set of experiments. The tournament algorithm, while robust for obtaining the maximum or top-\(k\) lists for a small integer \(k\) (smaller than the larger possible size of a ranking task we can assign to a human worker), cannot be applied to larger top-\(k\) lists. This is quite restrictive, as the size of individual human intelligence tasks is limited. We thereafter propose a recursive top-\(k\) approach that scales to any size of top-\(k\) list, making it applicable to a wider range of applications (e.g. for short-listing a set of tens of thousands college admission applications to a small set with hundreds). For this new class of algorithms that we propose, we describe an budgeting approach that allocates tasks across the stages of the algorithm in a way that addresses adversarial human behavior. We also propose a randomized algorithm for the recursive crowdsourced top-\(k\) algorithm. The randomized approach can admit a small risk of losing some items from the top-\(k\) list (a risk that is set by the user) in exchange for a very large reduction in the budget necessary to obtain the top-\(k\) list. Our experimental study validates the benefits of the proposed approaches, and shows that the methods provide high quality output with a low latency, even in very noisy environments of human workers, with a portion of workers having adversarial behavior.
and honest workers making mistakes at high rates.

Working without the presence of historical data, or with restricted knowledge for individual contributors as can happen in big, anonymous and transient crowds, we treat each datapoint as coming from a different user, and our techniques infer the quality of user input based on the nature of responses as opposed to other responses. With the use of fixed user identities, user reputation inference can become robust and enhance the quality of the output.

In Chapter 3, we propose novel algorithms for the problem of crowdsourcing binary labels, where users of known identities have been allocated with multiple tasks. Such binary labeling tasks are very common in crowdsourcing platforms, for instance, to judge the appropriateness of web content or to flag vandalism. We introduce two unsupervised algorithms: one derived heuristically which is simple to implement, and one based on iterated bayesian parameter estimation of user reputation models. We provide mathematical insight into the benefits of the proposed algorithms over existing approaches. We conduct a set of experiments on a multitude of different synthetic models and on a set of real-world datasets obtained through crowdsourcing platforms, and we validate these insights by showing that both algorithms offer improved performance on several occasions, compared to different approaches.

Bipartite graphs are a common occurrence in crowdsourcing settings as they naturally represent interactions between users on the one hand and the tasks they undertake on the other. The methods we describe in Chapter 3 work directly on bipartite graphs. Different settings can involve more generic graphs, e.g. interactions between
users on Wikipedia edits for the encyclopedia’s pages, form a generic graph. Learning on these graph structures is a challenge due to the irregular nature of the graph. Model-based methods such as Expectation-Maximization or the techniques we describe at Chapter 3, make a specific assumption on the underlying model describing the behavior of workers or the quality of items, and then work iteratively on learning the value of the parameters of the model. In doing so, they do not incorporate existing features that may be available on the interaction of users and items. In many cases, we can have access to a set of such informative features. Moreover, the apriori assumptions on the model of worker behavior may not accurately represent reality. In chapter 4, we introduce a machine learning approach that learns directly from the neighborhood of the graph. The method unfolds the graph at each node, creating a tree unfolding that serves as the training instance of a Multi-Level LSTM architecture. The Multi-Level LSTM architecture is trained using an adaptation of back-propagation (tailored to the multiple levels). We provide results for different settings that show the effectiveness of using this architecture for learning directly on the graph with minimal effort for feature engineering.
Chapter 2

Efficient techniques for Crowdsourced Top-k Lists

2.1 Problem setting and contributions

In this chapter, we introduce an algorithm to obtain the top-$k$ items out of a larger itemset by using the crowd to evaluate the “goodness” of items. An example instance of this problem is selecting a handful of the most “appealing” photographs out of a larger set. Another example is selecting the few most qualified candidates for a specific job out of a pool of applicants. In both of these cases, the quality of an item (a photograph or a job candidate) cannot be computed by an automated method. Instead, our algorithm assigns to the crowd the task of ranking a small number of such items. By carefully selecting these ranking tasks and combining their results, the algorithm generates the $k$ best items, as judged by the crowd. An underlying assumption here
is that the crowdsourcing service provides access to workers who can perform such rankings. Hence, we may use Amazon’s Mechanical Turk to rank photographs, but we will rely on a specialized service of experts to rank job applicants.

Our algorithm has to address several challenges that stem from the usage of crowdsourcing. Recruiting human workers and obtaining their answers incurs a high latency and may involve a monetary expense. In most crowdsourcing settings, there is a three-way tradeoff among quality of results, latency and monetary cost; optimization of one quantity comes in the expense of the two others. Some crowdsourcing applications are expected to return results at interactive speed. There are analytical methods for real-time crowdsourcing [5] that optimize latency. The problem that we consider may involve large itemsets and may require a large number of comparison tasks, therefore, in many cases we cannot realistically expect the results to be obtained in high speed. Our technique focuses on optimizing monetary cost and makes several calls to the service before obtaining the final results. It is thus not suitable for real-time applications, however, it keeps latency reasonably low by limiting the number of calls to the crowdsourcing service. The algorithm cannot simply employ large ranking tasks, as there is an inherent constraint on the size of a task. For instance, a worker may be able to rank up to a maximum of ten photographs at a time. Another challenge stems from the different judgements made by human workers on the same ranking task, which in turn introduces uncertainty in the ranking of items. The algorithm can aggregate answers for the same ranking task from several workers in order to reduce uncertainty, but this requires more calls to the crowdsourcing service and hence increases latency and expense.
Uncertainty also depends on the difficulty of ranking tasks, e.g., ranking photographs may be more straightforward than comparing applicant resumes. Moreover, workers may be more prone to errors depending on the crowdsourcing service. For instance, it is well known that Amazon Mechanical Turk has a considerable population of (hard to detect) spammer workers who intentionally provide false answers to tasks.

Previous studies have addressed specific variants of the top-$k$ problem. In [60], Venetis et al. explore algorithms for obtaining the maximum (i.e., top-1) item assuming some a-priori knowledge about the errors in the answers obtained from the crowdsourcing service. In contrast, our algorithm solves the generalized top-$k$ problem without requiring this knowledge. Another study [11] explored the more general top-$k$ problem assuming that workers rank two items at a time. Our algorithm does not have this restriction and can issue ranking tasks of several items each. Depending on the underlying domain, this strategy can reduce greatly the number of calls to the crowdsourcing service without compromising the accuracy of the final top-$k$ items. Relevant to the top-$k$ problem is the work in [2], which presents a way of sampling a quasilinear number of pair-wise comparison results for the purpose of learning to rank, but performs full sorting with no emphasis on the top-$k$. The proposed algorithms in [39] can be used to solve the top-$k$ problem by first obtaining the full sorted order and then returning the first $k$ items. However, this approach wastes money and time to perform uninteresting comparisons beyond the first $k$ items. This waste can be significant if $k$ is much smaller than the total number of items, which is common in practice. Also, existing proposals in the literature generally lack a robust defense mechanism against the problem of
spamming which is rampant in crowdsourcing [33]

The main idea of our algorithm is to set up a tournament to gradually reduce the input itemset down to the top-\(k\) items. Within each round of the tournament, the algorithm compares items by issuing several ranking tasks to the crowdsourcing service. The number of workers assigned to each task is chosen adaptively, based on the perceived complexity of computing a ranking. Detecting this complexity in a robust and adaptive fashion is one of the key technical challenges that we address in our work.

The tournament algorithm that we introduce has a limitation as it can obtain top-\(k\) lists only for \(k\)'s that are smaller than the size of the ranking tasks of human workers. Our work and relevant research on ranking tasks in crowdsourcing [39] has confirmed that there is an inherent limitation to the size of crowdsourced ranking tasks. On the other hand, the size \(k\) of a top-\(k\) list may be high. For example, we may query a database of millions of images for the top-100 list of images. The tournament algorithm would not scale, since we would have to ask human workers to provide rankings for sets of more than 100 images, a very large task for human computation. In the second part of this chapter, we introduce a recursive algorithm that maintains the size of the ranking tasks low, but can obtain top-\(k\) lists for any \(k\).

More concretely, the main contributions of our work can be summarized as follows:

- We define a tournament algorithm to solve the top-\(k\) problem that employs human workers to compare several items at a time (Section 2.3). The algorithm is parameterized by a component to aggregate the answers of several workers on the
same ranking task, and a component to adaptively select the number of workers for the ranking of a specific set of items.

- We introduce instantiations for the aforementioned components (Sections 2.3.2, 2.3.4 and 2.3.5) that work adaptively and do not require a-priori knowledge about the errors committed by human workers or the existence of spammer workers. The resulting top-$k$ algorithm can thus be deployed in different problem domains with minimal tuning.

- We describe a class of recursive algorithms that obtain top-$k$ lists from the crowd for arbitrarily large itemsets and large top-$k$ lists. The adaptive distribution of tasks of the tournament algorithm is applicable to the recursive algorithm. The algorithm issues comparison tasks whose total number is linear in the size of the input itemset, and the latency, measured as the number of roundtrips to the crowdsourcing service, is logarithmic in the size of the itemset.

- For the recursive top-$k$ algorithm, which is applicable to large top-$k$ lists and therefore needs a large number of tasks, we propose a budgeting strategy, that based on an analytic estimate of the impact of adversarial users to the output, distributes the available budget efficiently across the stages of the algorithm.

- We propose a randomized variant of the large top-$k$ algorithms that can reduce the required budget drastically by taking a negligible risk of compromising the quality of the output result.
• We present experimental studies for both the tournament algorithm for small top-
  k lists and the recursive algorithm for large top-k lists. We use both simulations
of human crowds and actual workers from Amazon Mechanical Turk. The results
for the tournament algorithm demonstrate that it computes accurate results even
if ranking becomes difficult or there are many spammer workers (up to 40% of the
worker population). Moreover, the algorithm matches the performance of a spe-
cialized competitor for the top-1 problem, and offers significant savings compared
to the state-of-the-art method that sorts the entire itemset. The experiments
of the recursive algorithm for large top-k lists allow us to draw conclusions on
the efficiency of the budgeting strategy, the randomized variant and the trade-off
among latency, cost and quality of results. Using equal budgets, the recursive
crowdsourced top-k algorithm provides higher quality output than unbalanced
rank estimation [62] and tournament algorithms ([48] and Section 2.3), while it
The randomized approach leads to significant budget saves, that can exceed 50%
with a very low risk of losing items of the top-k list. The randomized algorithm
is particularly beneficial for very large itemsets and large top-k lists.
2.2 Notations

2.2.1 Preliminaries.

We consider a set of items $O$ with cardinality $n$. A ranking of the itemset is a permutation of the $n$ items, where the first elements of the permutation are the highest ranked items and the last elements of the permutation are the lowest ranked items. Let $\sigma$ and $\tau$ denote two rankings. By $\sigma(i)$ we denote the rank of an item $i$ in ranking $\sigma$, that is, its position in the ranking. For the highest ranked element $t$ in $\sigma$, $\sigma(t) = 1$ and likewise for $\tau$. For two items $i$ and $j$ if $\sigma(i) < \sigma(j)$ we say that $i$ is ahead of or better than $j$ in ranking $\sigma$. Given an integer $k$, an item $i$ is a top-$k$ item in $\sigma$ iff $\sigma(i) \leq k$. A top-$k$ list is the set of all top-$k$ items. Note that, by the definition, a top-$k$ list is a set and not a ranking. That is, it does not capture the rank of the items within the list.

We assume the existence of a crowdsourcing service that allows a requester (in this case, our algorithm) to post tasks that can be completed by human workers. We consider a specific type of task defined as follows: Given a set of $s$ items, ask a worker for a ranking of the $s$ items according to the worker’s perception. We term this type of task a $s$-ranking task. Of course, the answers of different workers may differ for the same task, either due to differences in perception of item “goodness” or simply because workers may act as spammers who return random answers just to collect the reward. For this reason, we cannot assume any collective properties on the answers from different workers, e.g., that the returned rankings will define a total order. Note that the result of a task is the ranking of the $s$ items and not just the set of top-$k$ items. As
we discuss later, this requirement is crucial in order to ensure the robust aggregation of answers from different workers. We assume that each task is issued in a crowdsourcing service as a single Human Intelligence Task (HIT).

We associate two cost metrics with using a crowdsourcing service, namely latency and expense. We measure latency as the number of roundtrips to the crowdsourcing service, where a roundtrip involves the parallel issuing of several tasks and subsequently the collection of answers from the workers. We assume that there are enough workers to work the posted tasks in parallel, and hence it is desirable to issue many tasks in each roundtrip. Each roundtrip may be in the order of several hours, and so it is also desirable to reduce the total number of roundtrips. The expense metric involves the reward paid to workers for the completed tasks. We adopt the common practice of paying the same amount for each task, and hence we measure expense in terms of the total number of posted tasks, across all roundtrips.

2.2.2 The top-\( k \) problem.

We assume the existence of a ground-truth ranking \( \beta \) that sorts the items in \( O \) according to some property of interest, as judged by the workers of the specific crowdsourcing service. For instance, \( \beta \) represents the ranking of photographs based on their appeal or the ranking of job candidates based on their qualifications. Ranking \( \beta \) is unattainable for practical purposes, as it would require asking and aggregating the preference of all workers for all items in \( O \). It is not used by our algorithms; we only use it to reason about correctness. In our evaluation section, we evaluate accuracy of
results by comparing against a known “gold standard” ranking $\beta$.

Our goal is to compute $k$ items that are close to the top-$k$ items in the ideal ranking $\beta$, by asking a limited number of the aforementioned ranking tasks. Hence, the problem that we solve can be defined informally as follows: Given an itemset $O$ and a positive integer $Q$, compute a subset $\Omega$ of $O$ such that $\Omega$ contains $k$ items, the computation of $\Omega$ requires at most $Q$ ranking tasks, and the rank $\beta(i)$ of each item $i$ in $\Omega$ is close to the interval $[1, k]$. To define closeness formally, we reason in terms of an error measure. We assume every item $i$ has an associated real value $V(i)$, where for any item $j$ in $O$ where $\beta(i) < \beta(j), V(i) > V(j)$, that is, if the rank of $i$ in the baseline ranking is better than the rank of $j$, the value $V(i)$ of item $i$ is higher than the value of $j$. Let $e_1, ..., e_k$ be the top-$k$ items in $\Omega$ returned by the algorithm. We define the error measure $e_r = \frac{\sum_{i=1}^{k} V(\beta_i) - \sum_{i=1}^{k} V(e_i)}{V(\beta_1) - V(\beta_k) + 1}$, where $\beta_i$ denotes the item with rank $i$ in the baseline ranking $\beta$. Hence, the goal of the algorithm is to minimize the error $e_r$ of ranking $\Omega$; the lower the error the closer are the ranks of the items of $\Omega$ to the top-$k$ items of the baseline ranking, and when $\Omega$ is the same as the top-$k$ of the baseline ranking, the error is zero. The use of function $V$ to represent the qualities of values of items with real numbers provides us with a useful tool for reasoning about the algorithm’s performance, as the error measure does not depend only on the relative position in the baseline ranking of the returned items, but on the distance of the returned items’ value to those of the baseline ranking. As comparisons between items of very different quality are easier (and less expensive to obtain) than comparisons of similar quality items, the measure penalizes errors in easy comparisons more than errors in difficult comparisons.
We make the intuitive assumption that an algorithm can infer the relative position of items in $\beta$ by issuing a sufficient number of ranking tasks. Specifically, if several workers answer the same ranking task and agree that $i$ is before $j$ in their answers, then most likely $\beta(i) < \beta(j)$. Deciding when there is enough agreement among workers to infer this relationship is a key technical challenge that we have to solve. We assume that $k < s$, that is, $k$ is smaller than the largest possible ranking task that can be executed by a human worker.

It would be possible to produce a tighter problem definition (and one that is more amenable to analysis) by making specific assumptions about the distribution of errors in the answers of human workers. However, as noted in a previous study [61], this distribution depends heavily on the similarity of items in the unknown ranking $\beta$ and is hence very difficult to obtain in practice.

2.3 Top-$k$ Tournament Algorithm

We now introduce our algorithm [48] for the top-$k$ problem defined previously. The algorithm works in iterations, where in each iteration it issues several $s$-ranking tasks, processes the answers provided by workers and then prunes away items that are not likely to be among the top-$k$.

In what follows, we introduce the algorithm and then describe two of its key components: how to aggregate the answers of different workers, and how to determine the number of workers assigned to each task.
Input: Itemset $O$, integer $k$, integer $s$

Output: $k$ items in $O$

Data: $\text{candidate\_set}$: an itemset, $\text{partition\_set}$: a set of itemsets

1. $\text{candidate\_set} \leftarrow O$;
2. while $|\text{candidate\_set}| > k$ do
   3. $\text{partitions} \leftarrow \text{Partition}(\text{candidate\_set}, s)$;
   4. Mark each itemset $p$ in $\text{partitions}$ as incomplete;
   5. $\text{candidate\_set} \leftarrow \emptyset$;
   6. while $\text{partitions contain incomplete subsets}$ do
      7. foreach incomplete subset $p$ in $\text{partitions}$ do
         8. Post $s$-ranking tasks for $p$ // Section 2.3.3;
         9. $R \leftarrow$ answers of the $s$-ranking tasks for $p$;
         10. $r \leftarrow$ top-$k$ items for $p$ as aggregated from $R$ // Section 2.3.2;
      11. if $p$ is completed ($r$ passes test of Section 2.3.3) then
         12. $\text{candidate\_set} \leftarrow \text{candidate\_set} \cup r$
   13. return $\text{candidate\_set}$;

Figure 2.1: Tournament algorithm for the top-$k$ problem.
2.3.1 Algorithm Definition

Figure 2.1 shows the pseudocode of the algorithm. The algorithm receives as input the itemset $O$, the target number $k$ for the top items, and the size $s$ of the ranking tasks ($s > k$), and outputs $k$ items from $O$ that are deemed to be the top-$k$ items in $O$ based on the results of ranking tasks.

The algorithm maintains a variable $candidate\_set$ that comprises items from $O$ that are candidates for the top-$k$ output. Initially, this candidate set is the entire itemset $O$. The algorithm then proceeds in iterations, where in each iteration some items in $candidate\_set$ are pruned based on the results of $s$-ranking tasks. Specifically, each iteration initially partitions (in a random fashion) the items in $candidate\_set$ in disjoint subsets of size $s$. Each subset corresponds to a group of $s$-ranking tasks that will be posted to the crowdsourcing service\(^1\). Initially, each subset $p$ is marked as incomplete to denote that the algorithm has not yet computed the top-$k$ items in $p$. Subsequently, the algorithm posts $s$-ranking tasks for each incomplete subset $p$. The number of such tasks is determined based on the difficulty of comparing items in $p$. We discuss one method for this in Section 2.3.3. Note that the posting of $s$-ranking tasks for the incomplete subsets occurs in parallel, i.e., the algorithm does not wait for the answers for one subset before posting the tasks for another.

Once all the tasks have been posted, the algorithm gathers answers for each subset $p$ and aggregates them to determine the top-$k$ items in $p$. We discuss one pos-

---

\(^1\)At most one subset may contain fewer than $s$ items. If this subset has fewer than $k$ items, then no tasks need to be posted.
sible aggregation method in Section 2.3.2. We use the aggregation results to measure consensus among workers; we think of consensus as a predictor of correctness and in the case of high consensus we consider the results correct and mark $p$ as completed. The top-$k$ items of completed subsets are inserted in the candidate set for the next iteration. The determination of completeness is closely coupled with the method to determine the number of tasks for each subset $p$ and we discuss this further in Section 2.3.3.

The iteration continues until all subsets have been marked as completed. At that point, the candidate set of the next iteration comprises the top-$k$ items from each subset $p$, and hence the size of the candidate set has been reduced by a factor of $s/k$. When the number of candidate items is not greater than $s$, the algorithm issues one final set of $s$-ranking tasks and aggregates the answers to compute the top-$k$ output. Overall, the algorithm will perform $\log_{s/k}(|O|)$ iterations, where each iteration may comprise several roundtrips to the crowdsourcing service. A single roundtrip is delineated by the issuing of $s$-ranking tasks and the processing of their answers. Lines 7-12 in Figure 2.1 constitute a roundtrip and are executed in parallel for each incomplete subset $p$.

One observation is that the algorithm computes the correct top-$k$ items if workers always return correct answers. However, due to worker disagreements and errors, and the existence of spammers, this guarantee is not feasible in the general case. Intuitively, the algorithm is likely to return highly ranked items if we have robust methods to aggregate the noisy answers for the same subset $p$ and to allocate more tasks to difficult subsets $p$. In what follows, we describe our implementation of these two methods.
2.3.2 Aggregating Answers to s-Ranking Tasks

As mentioned, our method assigns the same ranking task to a number of different workers. Producing a single ranking out of multiple rankings over the same set of items is a process called rank aggregation, which we use in our algorithm. The optimal aggregated ranking is the one that minimizes its distance from all the rankings with respect to a specific ranking distance metric. There is a variety of binary distance metrics between rankings over the same itemset. The footrule distance between two rankings \( \sigma \) and \( \tau \), denoted \( F(\sigma, \tau) \) is the expression \( \sum |\sigma(i) - \tau(i)| \). The Kendall tau distance, also called bubble-sort distance, denoted by \( K(\sigma, \tau) \), is the number of item pairs \((i, j)\) such that item \( i \) is ahead of item \( j \) in only one of the two rankings, that is, the two items are in different order in the two rankings.

Using the footrule distance, we call an aggregated ranking \( \sigma \) footrule-optimal if it minimizes the expression \( \sum_i F(\sigma, \tau_i) \), where \( \tau_1, \ldots, \tau_m \) is the set of rankings that are aggregated, and \( i \) iterates over all items in the rankings. Computing a footrule-optimal aggregation is tractable. However, computing a Kendall-optimal aggregation is equivalent to the minimum feedback arc set problem which is NP-hard. An early result by Diaconis and Graham [15] proved that the Kendall Tau distance and the Spearman’s footrule distance are ‘equivalent’, since they are within a factor of 2 from each other.

A heuristic for aggregating rankings is the median rank aggregation which we used to implement the aggregation of multiple rankings of the same partition. Median rank aggregation identifies the median rank of each item, i.e. for a set of rankings
\( \tau_1, \ldots, \tau_m \), \( \text{medrank}(i) = \text{median}(\tau_1(i), \ldots, \tau_m(i)) \). It then sorts the median ranks to produce a new ranking, so the rank of every item is the position of its median rank in the sorted list. As shown in [18], the median rank aggregation method gives a footrule-optimal aggregation in many cases. In [17], median rank aggregation was proven to be optimal, within constant factor of 3, for rankings that may involve ties, while the constant factor for rankings without ties was proven to be even lower at 2. For the pseudocode of median rank aggregation refer to Figure 2.3.2.

An algorithm to obtain the median rank aggregation is to access each of the rankings one element at a time, starting from the top elements of the rankings, until some item is seen \( \lceil \frac{m+1}{2} \rceil \) times, this is the winner item. A tie can be maintained so we may get more than one winners, or the ties may be broken arbitrarily. The algorithm can continue like this and get the runner-up item, the third item and so on. This algorithm was shown in [19] to be instance-optimal among the algorithms that access the input rankings sequentially, it is the best possible algorithm, within constant factor of 3, for any possible instance. Considering the rankings as vertical lists put one aside the other, we say that we access the first line of the rankings when we examine the top-1 elements in all the rankings, and likewise for elements of lower rank. In other words, the term line is used to refer to the rank, common for of all the rankings, that the median rank algorithm is examining as it accesses the rankings. We call the number of lines that need to be accessed to aggregate a number of rankings to obtain a top-k ranked list the depth of probe of the aggregation.

Given a subset of items \( p \), the goal is to aggregate several answers for the \( s- \)
Input: A set $R$ of $m$ rankings over itemset $S$, integer $k$

Output: A top-$k$ ranked list of items in $p$

Data: List: a ranking of items in $p$, topk: a top-$k$ ranked list

1. $List \leftarrow$ a random permutation of $p$;
2. foreach item $i$ in $p$ do Compute median rank of $i$ in rankings $R$;
3. Sort $List$ according to median ranks of items;
4. $topk \leftarrow$ first $k$ items in $List$;
5. return $topk$;

Figure 2.2: aggregate_rankings($R,k$) using median rank aggregation

As shown in [16], the aggregated ranking $\tau$ has several desirable properties. Intuitively, $\tau$ can be seen as an “averaged” ranking that minimizes the average distance to the initial rankings $\tau_1, \ldots, \tau_m$. We can thus treat $\tau$ as the consensus ranking that arises from the answers of different workers, and the first $k$ items in $\tau$ as the most likely
top-\(k\) items in \(p\). We establish the following result for median rank aggregations:

**Theorem 2.3.1.** The lowest possible median rank of a top-\(k\) item in the output of a median rank aggregation of \(m\) rankings over a set of \(s\) items is:

\[
L \geq \left\lceil \frac{(q-1)s + (k-1)(m-q+1) + 1}{m} \right\rceil, \quad q = \left\lfloor \frac{m+1}{2} \right\rfloor
\]

**Proof.** We use the median rank algorithm that accesses the rankings sequentially, starting from the top ranked items and continuing deeper, to perform the aggregation. We consider an adversarial scenario where the workers are delaying the completion of the aggregation as much as possible. For \(m\) rankings, median rank aggregation admits an item to the aggregated result when it encounters it \(q = \left\lfloor \frac{m+1}{2} \right\rfloor\) times. The median rank of an item that is admitted to the top-\(k\) ranked list at a line, is the number of that line and therefore the lowest median rank of an item in the top-\(k\) ranked list is the depth of the probe. We will establish the maximum depth of probe of the aggregation. For the winning top-1 item the workers can delay the aggregation by presenting all \(p\) items exactly \(q - 1\) times. After that, some item is encountered \(q\) times. Therefore, for a top-1 list the maximum number of lines to complete the aggregation is \(\left\lceil \frac{(q-1)p + 1}{m} \right\rceil\). The workers can continue presenting the same winner item for an additional \(m - q\) times, delaying the admission of the top-2 item, after which the winner item is exhausted. Any item that the workers present after that will be admitted as the top-2 element in the list. As before, the workers can delay the admission of the next item by presenting the previously admitted item an additional \(m - q\) times. Therefore, a top-k list will be complete after at most \(\lceil (q - 1)p + (k - 1)(m - q + 1) + 1 \rceil\) item presentations from
the workers, and since every line contains \( m \) items, the maximum depth of probe, and equivalently, the lowest median rank of an item in the top-\( k \) ranked list is in the worst case \( \left\lceil \frac{(q-1)p+(k-1)(m-q+1)+1}{m} \right\rceil \).

The significance of the theorem is that, when using median rank aggregation, it is sufficient to ask workers only for a ranking of the top-\( L \) items of each subset instead of a ranking of all \( s \) items, which can require less effort by the workers.

### 2.3.3 Determining the Number of Tasks

**Basic approach.** A simple way is to allocate the same number of workers to all \( s \)-ranking tasks and to vacuously consider the results of all aggregations as complete. We call this strategy *basic*. This way we minimize the number of roundtrips but the algorithm likely overspends HITs on easy tasks and spends less than required on hard tasks.

**Adaptive approach.** We propose an adaptive approach that uses a method to judge whether a subset is incomplete and determines the number of tasks posted for each subset at every roundtrip. By requesting a larger number of additional answers, the algorithm may finalize faster the top-\( k \) items since consensus may be more likely. However, this also means a higher expense. On the flip side, by requesting fewer additional answers at each iteration, the algorithm can spend resources more judiciously but it will also increase latency since additional answers come at the cost of a roundtrip. For the time being, suppose that we have a reliable method to detect whether the result of median-rank aggregation (see previous section) has high certainty and hence we can
mark a subset $p$ as complete. Let $\psi_1, \ldots, \psi_n$ be a sequence of positive integers such that $\psi_1 < \cdots < \psi_n$, where $n$ is a parameter of our scheme. We initially post $\psi_1$ tasks per subset $p$, where $\psi_1$ represents the minimum count of human workers whose answers can hopefully provide a consensus in median-rank aggregation. We found that $\psi_1 = 3$ is a reasonable default. If additional answers are required, then we post an additional set of $\psi_2 - \psi_1$ tasks, which means that the algorithm will have a total of $\psi_2$ answers to evaluate the completeness of $p$. We continue in the same fashion until we post a total of $\psi_n$ tasks which is the maximum. At that point, $p$ is marked as complete by default.

In practice, we found that it works well to have three levels $\psi_1, \psi_2, \psi_3$ ($n = 3$) corresponding to easy, medium and high difficulty, with $\psi_1 = 3$ and $\psi_2 = \lfloor \psi_3 + \psi_1 / 2 \rfloor$.

The number of subsets at each iteration $i$ is: $p_i = \lfloor |C_i| \div s \rfloor + d_i$, where $C_i$ is the candidate set at iteration $i$ and $d_i$ is 1 if $(|C_i| \mod s) > k$ and 0 otherwise. For $i > 1$: $|C_i| = (p_{i-1}) \cdot k + (1 - d_{i-1}) \cdot (|C_{i-1}| \mod s)$. Starting with $C_1 = O$, we can compute all $p_i$'s until the last iteration and obtain the total number of subsets. Since the total number of posted tasks cannot exceed $Q$, we define $\psi_3 = \lfloor \frac{Q \cdot \Sigma_i p_i}{s} \rfloor$.

The only remaining question is how to judge completeness for a subset $p$. In other words, given the answers to the $s$-ranking tasks for $p$ and the $k$ items resulting from median-rank aggregation, how can we determine that they represent the top-$k$ items with high certainty. Here we rely on the following intuition: If workers tend to disagree on their answers, this likely implies existence of spammers or difficult comparisons and allocating more workers to the task is likely beneficial in terms of quality. On the other hand, agreement among workers can imply an easy task, and therefore accurate results,
in which case we mark the s-ranking task as complete.

To measure agreement among the answers of workers, we resort again to the mechanics of median-rank aggregation from Section 2.3.2. We describe our technique with an example. Suppose that \( p = \{A, B, C, D, E\} \), \( k = 3 \), and the answers we obtain from three workers are as follows: \( \tau_1 = (A, B, C, D, E) \), \( \tau_2 = (A, C, B, D, E) \) and \( \tau_3 = (A, B, D, C, E) \). Median-rank aggregation will assign the following median-rank \( r(i) \) to each item \( i \): \( r(A) = 1 \), \( r(B) = 2 \), \( r(C) = 3 \), \( r(D) = 4 \), \( r(E) = 5 \), and hence the top-3 items are \( \{A, B, C\} \). Note that the median ranks of the top-3 items are the same as if the workers were in perfect agreement, i.e., they all returned \( (A, B, C) \) as their answer. Indeed, one can observe that there is high agreement in their answers, even though items may be ordered differently. This is reflected in the number of items that are assigned each median rank: there is exactly one item \( (A) \) with median rank 1, exactly one item \( (B) \) with median rank 2, and so on. In contrast, suppose that the returned answers corresponded to disagreement, as follows: \( (A, D, C, B, E) \), \( (C, A, B, D, E) \), \( (B, E, A, C, D) \). Here, the median ranks are as follows: \( r(A) = 2 \), \( r(B) = 3 \), \( r(C) = 3 \), \( r(D) = 4 \), \( r(E) = 5 \). The top-\( k \) items are again \( \{A, B, C\} \), but among them no item is given median rank equal to one, and two items have median rank equal to three. Overall, the intuition is that we can measure worker agreement by examining the ties in the assigned median ranks.

Formally, let \( r \) denote a vector of length \( L \), where \( L \) is defined as the lowest median-rank that can be assigned to the \( k \) items in the output of median-rank aggregation (Theorem 2.3.1). The component \( r[i] \) denotes the count of items (among the
$k$ output items of median-rank aggregation) that are assigned median rank $i$. In the case of perfect agreement among workers, we can show that $r[i] = 1$ for $1 \leq i \leq k$, and $r[i] = 0$ for $k < i \leq L$ (see also our example). Conversely, disagreement results in $r[i] = 0$ for some $i \in [1, k]$, or $r[j] > 1$ for some $j \in [1, L]$. Using this idea, we measure the uncertainty of median-rank aggregation by the distance between $r$ and the reference vector that represents perfect agreement. If this distance exceeds a threshold $t$ then we infer that workers disagree too much to infer the top-$k$ items with high certainty, and hence we mark $p$ as incomplete. We experimented with several distance metrics and settled on cosine similarity because of its robustness in our experiments. Threshold $t$ can be quantized to a few values, each one corresponding to a specific sensitivity to disagreements.

2.3.4 Speculative optimization for reduced latency

Since the cardinality of the candidate set decreases in every round, we expect the problem to get more difficult for the workers as we approach the final round, because the items that remain in the pool are likely more similar and harder to rank than items in the initial round that come from a larger and more diverse pool.

Thus, we may end up initiating extra roundtrips for every round as the tournament approaches the last rounds. In the worst case all partitions in a round may end up being allocated to a total of $\psi_{\text{max}}$ workers, incurring an extra latency of 2 roundtrips for the round. If we had prior knowledge that the round will be difficult, we could avoid the extra latency by issuing $\psi_{\text{max}}$ HITs per aggregation in a single roundtrip from the
beginning of the round.

To avoid unnecessary latency, we propose the use of an adaptive starting point \( \xi \), where each round starts by allocating to all aggregations the number of HITs \( \psi_i \) that was allocated to the majority of aggregations in the previous round. If there is a tie, \( \xi \) is set to be the largest of the levels.

If the majority of aggregations in the previous round was completed using \( \psi_{\text{min}} \) workers, the round starts with \( \xi = \psi_{\text{min}} \) and the optimization can initiate up to 2 extra roundtrips.

If there was a large number of optimizations and the majority of aggregations was completed using \( \psi_{\text{med}} \) HITs, the next round starts by allocating \( xi = \psi_{\text{med}} \) HITs to all aggregations. We give aggregations that test positive for optimization to a total of \( \psi_{\text{max}} \) workers. Thus, only one extra roundtrip is possible in this case. If the starting point becomes \( \psi_{\text{max}} \), no further optimization is possible.

We call this optimization speculative as it assumes that we will need more workers without checking it.

The technique may slightly increase the number of HITs by allocating extra HITs to some easy aggregations, but for difficult tournaments we expect the number of roundtrips to be significantly lower, maintaining the same quality.

2.3.5 Redistributive optimization

We calculate levels \( \psi_{\text{max}} \) and \( \psi_{\text{med}} \) in the beginning of the tournament using the given budget \( Q \). They remain constant until the completion of the tournament.
However, in some rounds there may be none or very few optimizations, leaving a margin for a larger $\psi_{max}$ in subsequent rounds, since the remaining number of aggregations is lower.

We can thus recalculate $\psi_{max}$ after each round to be the largest integer for which:

$$\psi_{max} \cdot \#aggr(Candidate\ Set) \leq Q - (\#used\ HITs),$$

where $\#aggr(S)$ is the number of aggregations that are required to get the top-k list out of itemset $S$. After recalculating $\psi_{max}$ we also recalculate $\psi_{med}$ using the formula $\psi_{med} = \frac{\psi_{max} - \psi_{min}}{2}$ as before.

This adaptation can lead to a more efficient usage of the budget, by increasing $\psi_{max}$ if there are few optimizations in the initial rounds. Thus, we can give the difficult comparison tasks of the last rounds to a substantially increased number of workers. Note that this optimization can be applied concurrently with the speculative optimization.

### 2.4 Experimental study of the tournament algorithm

#### 2.4.1 Simulations

**Data.** We generated synthetic itemsets of varying size $n=50, 100, 200, 300$. Every item $i$ in $O$ is associated with a real value $V(i)$ that determines a ground-truth ranking $\beta$ for the elements—a high value implies a high rank and vice versa. We generated three itemsets corresponding to different methods of assigning values to items: the first itemset assigns values uniformly at random in the interval $[1, n]$; the second itemset
assigns integer values 1, 2, \ldots, n; and, the third dataset assigns values from a Gaussian distribution with mean $\lceil \frac{n+1}{2} \rceil$ and deviation $\frac{n}{4}$, meaning that on average more than 98% of items take values in the [1,n] range with the majority centered around the mean.

Our results for the third value distribution are better than for the other two and seem to favor our algorithm, since it is easier to differentiate the top-k items. We therefore omit this itemset from the presented results. The results for the uniform-random and integer-valued itemsets were practically the same, thus, we consider only the simplistic integer-valued itemset as it makes it easier to interpret our experimental results.

**Algorithms.** We implemented both the basic and adaptive approaches and compare them. Unless otherwise noted, the adaptive method allocates any remaining HITs (up to the specified budget $Q$) to the very last $s$-aggregation that also determines the top-$k$ items. We also consider the extension of the adaptive approach with the speculative and redistributive heuristics.

**Modeling of human workers.** A human worker may be either an honest worker or a spammer, with a probability that we vary in our experiments. A spammer returns any of the $s!$ permutations of the items. An honest worker behaves according to the model that Louis Thurston described in 1924 [59], formalizing the fuzzy way that humans perceive intensity of physical stimuli. Specifically, we assume that the worker perceives the value of an item $i$ by sampling from a normal distribution that is centered around the actual value $V(i)$. Figure 2.3 gives an example of this process, assuming that the worker is presented with three items to rank. The perceived values determine the ranking returned by the worker. Clearly, the probability of committing errors (with
Figure 2.3: Example of a swap in the baseline ranking. Each worker perceives a distorted subjective view of the true ranking of an item by sampling from a Gaussian distribution with the true value of the item as mean, and a variance which corresponds to the level of accuracy of the worker. Swaps can happen due to the distortion. This model is based on psychometric studies on how human subjects perceive natural stimuli [59] and has been used in relevant crowdsourcing literature for modelling errors by workers [60].

respect to the ground-truth ranking) increases with the variance of the distribution.

We treat the top-\(k\) items as a set, and so swaps that take place within the top-\(k\) set are irrelevant and do not affect our metrics. Hence, we model the error rate of honest workers as the probability that there is a swap of two items of value distance equal to \(k\). This probability corresponds to a specific variance of the normal distribution of the error model. As an example, an error rate of 30\% for \(k=5\) corresponds to a probability of more than 49\% in swapping two neighboring items in a ranking task. We assume that all honest workers make errors with the same rate, that is, we do not assume the existence of workers with higher or lower expertise than others.

**Parameters.** We fix the size of the ranking tasks at 10, the threshold beyond which workers abandon tasks at a high rate, as explained in [39]. The following table shows the remaining parameters of the experimental study, how we vary them and their default values.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Range</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k$</td>
<td>1,5</td>
<td>5</td>
</tr>
<tr>
<td>$s$ (size of ranking task)</td>
<td>-</td>
<td>10</td>
</tr>
<tr>
<td>Workers per ranking task</td>
<td>3-15</td>
<td>3</td>
</tr>
<tr>
<td>Spammer prob.</td>
<td>10%-60%</td>
<td>20%</td>
</tr>
<tr>
<td>Error rate</td>
<td>10%-40%</td>
<td>25%</td>
</tr>
<tr>
<td>Budget</td>
<td>198-10000</td>
<td>585</td>
</tr>
<tr>
<td>Threshold $t$</td>
<td>-</td>
<td>0.85</td>
</tr>
</tbody>
</table>

**Error measure.** Let $e_1, \ldots, e_k$ be the top-$k$ items returned by the algorithm. We define the error measure $\epsilon_r = \frac{\sum_{i=1}^{k} V(\beta_i) - \sum_{i=1}^{k} V(e_i)}{V(\beta_1) - V(\beta_k) + 1}$ where $\beta_1, \ldots, \beta_k$ are the top items in the ground-truth ranking, as we described in Section 2.2.2. That error measure is based on values instead of ranks so as to take into account the similarity among the returned top-$k$ items and the actual top-$k$ items. Note that the metric acquires a natural interpretation for the integer-valued itemset we use in the experiments, where items with neighboring ranks in $\beta$ have a value distance equal to one. Hence, the factor $V(\beta_1) - V(\beta_k) + 1$ is equal to $k$ and the formula expresses the average value error per item in the top-$k$ list. As an example, for $k = 5$, $V(\beta_1) = 100$ and $V(\beta_5) = 96$, we are essentially normalizing by $k$ ($=5$) and the error metric can be interpreted as the loss per each item in the top-5 list. An error of, say, $\epsilon_r = 4$ means that each item we obtain in the returned top-5 list is valued on average 4 units less than the correct (and since the value distances are 1, also 4 ranks lower). Normalizing by the range of differences scales down the error metric, for non-equidistant value distributions. Normalizing by the sum would have analogous results (it does not affect the comparative performance, as it would only normalize by a different constant number), but the interpretation of the error measure as value loss per item would not hold for the equidistant assumption.
Figure 2.4: Performance for increasing budget for obtaining the top-5 list of a set of 200 items. We see comparative performance for the basic tournament, the tournament using the adaptive task allocation technique, the adaptive allocation technique together with the speculative optimization, and the adaptive allocation technique together with both the speculative and redistributive optimizations. Both the percentage of random spammers and rate of honest worker mistakes are at the default level.

For every arrangement of the parameters we conducted 50 experiments (using different seeds for our random number generation) so that the average error value converges to its actual expected value. The error we report in the plots is the average error $\epsilon_r$ over all 50 runs.

**Basic vs. adaptive approach.** Figures 2.4 shows the performance of the algorithm for three levels of available budget, namely 198, 351, 585, corresponding to 3, 9 and 15 tasks respectively per subset of $s$ items for the basic algorithm, for default error rates by workers in the crowd. Figure 2.5 shows results for a similar setting where the level of error rate by honest workers is at 40% modeling a scenario where the comparison tasks are more difficult. We observe that in both figures the adaptive variant reduces error significantly (between 1.5x and 2x) compared to the basic variant. The reason is the adaptive allocation of workers depending on the difficulty of ranking tasks, which
Figure 2.5: Performance for increasing budget for obtaining the top-5 list of a set of 200 items. We see comparative performance for the basic tournament, the tournament using the adaptive task allocation technique, the adaptive allocation technique together with the speculative optimization, and the adaptive allocation technique together with both the speculative and redistributive optimizations. The percentage of random spammers is set to the default level while the rate of honest worker mistakes is set to 40%, which models a scenario where the comparison tasks are very hard.

increases in the later iterations. We also see that the speculative heuristic can reduce latency significantly, cutting down the number of roundtrips by 44%. On the other hand, the redistribution heuristic performs worse than expected and results in a higher error. The reason is that the problem becomes dramatically more difficult in the last iteration, and hence the algorithm can benefit more from keeping the $\psi_i$ levels lower in the initial iterations, reserving an ample budget for the last iteration. Given these results, from now on we focus on the adaptive variant augmented with the speculative heuristic.

**Tolerance to workers errors.** Figure 2.6 shows the error measure as a function of the error rate of honest workers. The two curves correspond to no spammers and to the default probability of getting a spammer worker. As expected, the output error
Figure 2.6: Error levels of the tournament algorithm using the adaptive task allocation technique and the speculative optimization, for different levels of the error rate of honest workers. One curve shows performance for when random spammers are absent, while the second curve shows performance for the default level of spammers. As expected, the output error increases as the workers commit more errors, but the algorithm can achieve high accuracy even for very high error rates. For example, for an error rate of 30%, which corresponds to a probability of more than 49% of swap for items with neighboring quality value, the algorithm returns the top-k list with average error less than 0.2.

increases as the workers commit more errors. However, we observe that the algorithm can achieve high accuracy even for high error rates. For instance, for an error rate of 30%, which corresponds to a probability of more than 49% of swap for neighboring items, the algorithm returns the top-k list with average error less than 0.2, which corresponds to returning elements $\beta_1, \ldots, \beta_{k-1}, \beta_{k+1}$, i.e., missing only $\beta_k$ and substituting it with the next best element $\beta_{k+1}$.

**Tolerance to spammers.** Figure 2.7 shows the error measure as a function of the probability of having a spammer worker answer an s-ranking task. The two curves correspond to the default error rate and to an error rate of zero, i.e., honest workers that do not commit errors. The results demonstrate that the algorithm is highly resilient
Figure 2.7: Error levels of the tournament algorithm using the adaptive task allocation technique for different levels of random spammers in the crowd. One curve shows performance for a setting where there are no errors by honest workers, and the second curve shows results for a setting where there is the default error rate of human workers. The results demonstrate that the algorithm is highly resilient to spammers, with a low error even for a very high percentage of spammers. The adaptive task allocation technique detects disagreement and posts more tasks as needed to improve the quality of contentious ranking tasks.

to spammers, with a low error even for unrealistically high percentage of spammers (e.g., more than 40%). The key observation here is the algorithm can battle spamming effectively by detecting disagreements and posting more tasks as needed.

Comparison to human-powered sorts. We implemented the Compare operator proposed in [39] and used it to obtain the top-k items by performing a full sort of $O$. Figure 2.8 compares our algorithm with Compare for two possible allocations of HITs to the operator. Our algorithm outperforms Compare in terms of accuracy and total number of HITs. Specifically, to match the accuracy of our algorithm, Compare requires an order of magnitude more HITs. When allotted twice as many HITs as our algorithm, Compare has an error that is 6x higher. The reason is that Compare computes a full ranking of the items, which is clearly wasteful for our problem.
Figure 2.8: Comparative performance of the tournament algorithm with the *Compare* operator proposed in [39]. To batch the 300 items of the set into HITs, we use a batching algorithm that occasionally produces overlapping batches. The tournament algorithm outperforms *Compare* in terms of accuracy and total number of HITs, which by issuing a quadratic number of tasks is wasteful for our problem setting.

**Comparison to max tournament.** We implemented the tournament max algorithm presented in [60] for obtaining the top-1 item. We tuned tournament-max to the average behavior of our users, which includes both spammers and honest workers who commit mistakes. Figure 2.9 shows the error measure for tournament-max and our proposed adaptive approach. As shown, the performance of our algorithm is comparable, yet the difference is that we do not require any a-priori knowledge about the behavior of human workers. Moreover, our technique can work for $k > 1$.

### 2.4.2 Experiments over Mechanical Turk

**Data.** We tested three types of itemsets of size 160. For $k = 5$ and $s = 10$, this size ensures that all comparison tasks will be exactly $s$ in size, and the tournament will terminate in 5 iterations, decreasing the size of the candidate set to 80, 40, 20,
Figure 2.9: Comparative performance of our adaptive tournament algorithm against the tournament max algorithm presented in [60] for obtaining the top-1 item. We tuned tournament-max to the average behavior of our users, which includes both spammers and honest workers who commit mistakes. The performance is comparable, but without the need for a-priori knowledge about the behavior of workers.

10 and 5 respectively. For all generated itemsets, the items are shapes and the goal is to find the top-\(k\) items with the largest area. The first itemset comprises squares whose edge lengths range from 20 to 180. The difference in size among squares is easy to discern, therefore we consider this itemset as modeling an easy case. The second itemset comprises polygons of varying area and varying vertex count (between 4 and 10). The difference in shapes makes it harder to discern differences in area, and hence we consider this a medium-difficulty itemset. Finally, we create a high-difficulty itemset by reducing the difference in area size among polygons.

In Figure 2.10 we see an example of a completed ranking tasks of size 10. This example was located at the preamble of the HIT HTML form that was presented to workers of Amazon to make the task clear for them. In Figure 2.11, we see the actual task where the human worker is prompted to rank the 10 polygons.
Rank the blue polygons according to their size (largest to smallest)

At the bottom of the page you see 10 blue regular polygons of different sizes. Your task is to compare the sizes of the polygons and rank each polygon according to its size, i.e. the area that the polygon covers.

The rank of the largest polygon is 1 and the rank of the smallest polygon is 10.

To determine the rank of a polygon one should observe its size and compare it to all other polygons.

An example of a correct assignment is the one you can see below. The polygons are ranked according to their size:

As in the above example, write the rank number, ranging between 1 and 10, in the textbox below each blue polygon of the task below. The comparison result between two polygons may be obvious, or it may be difficult to determine which is ranked higher.

The task can be tricky because the polygons have varying numbers of vertices, e.g. one may be a square and another one may be a hexagon. All polygons have different sizes and cover different areas and there are no ties.

Figure 2.10: An example of a completed ranking task of 10 polygons, that we provided to the human workers for illustrative purposes at the preamble of the HTML form of the Human Intelligence Tasks.
Figure 2.11: An actual ranking task of a set of 10 polygons that human workers had to complete as part of our Mechanical Turk experiments.

**Workers.** We used the workers of Amazon’s Mechanical Turk without imposing any restrictions nor requiring qualification tests. We paid $0.02 per task. We did not allow our algorithm to exhaust the budget (600 HITs) in every experiment, instead, we were more conservative in the last iteration choosing to continue posting HITs only if the algorithm kept identifying an $s$-ranking task as incomplete.

**Error measure.** The error measure is the same as for the simulations, using the normalized area of each item as the value (that is, the area of the item divided by the difference in area between neighboring items). The results we report for each case are the average error from 3 identical experiments.

**Results.** Figure 2.12 shows the error measure for our algorithm as a function of the difficulty of the itemset. For comparison, we also plot the basic variant and the
Figure 2.12: Comparative performance of the basic variant of the tournament algorithm, against the variant using the adaptive task allocation technique, for a set of 160 shapes with the task of obtaining the top-5 list using tanking tasks of size 10. For comparison we also experimented with the technique using the speculative optimization. The adaptive task allocation achieves a significant boost in performance by lowering the error for all three levels of difficulty of comparisons.

adapative variant without the speculative heuristic. The trends validate our simulation results, showing that the adaptive variant yields a substantial drop in the overall error in all cases. The errors increase sharply for the difficult itemset, but the actual ranking task is extremely challenging due to the very small differences in area among shapes. (In some sense, we made the task a bit too difficult for human workers.)

2.5 Large top-$k$ lists

As we saw, a natural approach to the top-$k$ problem is tournament style algorithms, as in [60] which obtains maxima and the approach we describe at section 2.3 which obtains small top-$k$ lists. As the experimental results above show, the method we introduce in section 2.3 has comparable performance to the methods in [60] for
the max problem and employs a technique that addresses random spamming but not vandalism, i.e. adversarial spamming by workers who invert the correctness of results. The distinguishing characteristics of our approach are that it allows workers to examine several items at a time, it does not require any a-priori knowledge about the errors of human workers, and it adapts dynamically to the varying difficulty of comparing items and the existence of spammers. The experimental results above demonstrated that our algorithm yields accurate results, even when honest workers are likely to make mistakes and there is a large number of spammer workers. The tournament approach is restricted only to small top-$k$ lists where the size of the ranking tasks is larger than the size of $k$. On the other hand, the size $k$ of a top-$k$ list may be high. For example, we may query a database of millions of images for the top-100 list of images. The tournament algorithm would not scale, since we would have to ask human workers to provide rankings for sets of more than 100 images, a very large task for human computation. The method studied in [11] is a randomized tournament approach in the initial stages of execution. When this method obtains a reduced enough candidate set, it invokes the method described in [20] to obtain the top-$k$ list. The method in [20] has a very high latency, since it is essentially a heapsort algorithm for noisy comparisons, where comparisons take place sequentially and not in parallel. Thus, the technique in [11] is also of high latency. Moreover, the method’s analytic results hold under specific assumptions for the error functions of human workers. In practice, it is difficult to have any knowledge on worker error distribution a priori [61]. Also, the study does not consider the case of adversarial human workers, which are a major challenge in human computation. Worker tasks are
restricted to pairwise comparisons, while in reality human workers can perform tasks containing significantly more than two items. In section 2.6, we introduce techniques that do not have this restriction. The work in [9] assumes prior knowledge on the quality of the items, which is not realistic for many applications. A randomized sorting algorithm was studied in [62], in which predicted permutations are more accurate near the top rather than the bottom of the sorted list. In our experimental study, we compare against this approach.

### 2.6 A Recursive Crowdsourced Top-$k$ Algorithm

We introduce an algorithm [13] for the top-$k$ problem that leverages the knowledge from comparisons that the crowd has completed to decrease the set of candidate items of the top-$k$ list rapidly by pruning away items that are not likely to be in the top-$k$ list.

The algorithm decouples the size of the human comparison tasks from the size of the top-$k$ list and can output results for arbitrarily large top-$k$ lists.

In this section, we describe the algorithm and reason about its correctness in obtaining the top-$k$ list of a larger itemset assuming that human workers provide correct answers, i.e. answers that are consistent with the baseline ranking. In reality, human workers can return incorrect results and lead to errors in the output of the algorithm. In the section *Handling inaccuracy of crowds* we describe techniques that make the algorithm robust against inaccurate answers of the crowd.
Figure 2.13 shows the pseudocode of the algorithm that we call Crowd-Top-$k$. Its input is an instance of the top-$k$ problem. The algorithm has two stages: the reduction stage and the endgame stage.

We assume that the ranking tasks issued to the crowd have a limited size $s$. If the itemset $I$ is large enough such that it can be partitioned into more than $k$ partitions ($|I|/s > k$), the algorithm is in the reduction stage. It partitions $I$ and obtains rankings through crowdsourcing. Once the method gets the rankings of the partitions through crowdsourcing, it picks the maxima, i.e. the top-1 items of each partition, and forms a new itemset $\hat{I}_m$. It then calls itself to obtain the top-$k$ list of the itemset $\hat{I}_m$, i.e. the top-$k$ list of the maxima from all the partitions of $I$. Thereafter, it uses the knowledge from the top-$k$ ranked list of the maxima (structure $\hat{T}_m$) and the rankings over the subsets of $I$, to make comparison inferences and prune away items that are not candidates for being in the top-$k$ list of $I$. Every partition of $I$ whose maximum item is not in the top-$k$ list of $\hat{I}_m$ can be discarded. This is due to transitivity of comparison results which are consistent with the baseline ranking; the rank of any item in such a partition is at most the rank of its maximum, yet, the maximum has lost to more than $k$ items. Thus, no item of the partition can be in the final top-$k$ list. For each partition of $I$ whose maximum belongs to the top-$k$ ranked list of $\hat{I}_m$, assuming correct comparisons, we can infer the following: the final rank of an item of the partition is at least the rank of the maximum of the partition in the top-$k$ ranked list $\hat{T}_m$ of $\hat{I}_m$ plus the rank of the item in the partition. For items with rank that can still be less than $k$, that is, their rank in the partition augmented by the rank of the partition’s maximum item in $\hat{T}_m$ is less or
Crowd-Top-k(I,k,s)

Input : Itemset I, integer k, integer s (size of partitions)

Output: Top-k ranked list of items in I

if \(|I|/s \leq k\) then
    return endgameTopk(I, k, s);

Partition I in subsets\{S_1, ..., S_{|I|/s}\} ;

Obtain full ranking \(R_i\) of each \(S_i\) through crowdsourcing;

\(\hat{I}_m \leftarrow\) set of max items from all subsets \(S_i\);

\(\hat{T}_m \leftarrow\) Crowd-Top-k(\(\hat{I}_m, k, s\));

\(C \leftarrow\) \emptyset;

foreach item i in \(\hat{T}_m\) do
    \(C \leftarrow C \cup i;\)
    \(R \leftarrow\) ranking \(R_v\) where \(i\) belongs to \(R_v;\) //obtained in line 5
    foreach item j in R different than i do
        if \(((\hat{T}_m(i) + R(j)) \leq k)\) then
            \(C \leftarrow C \cup j;\)

return endgameTopk(C, k, s);

Figure 2.13: Recursive algorithm Crowd-Top-k for the top-k problem
equal than k, stay in the game and form part of the new set of candidate items for the top-k list of I (set C).

The new set of candidate items \( C \) contains \( 1 + 2 + \ldots + k = \frac{k(k+1)}{2} \) items if \( k < s \), and \( (k - s) \cdot s + \sum_{i=k-s+1}^{k} k - i + 1 \) for \( k \geq s \), which in both cases is \( O(k \cdot s) \). The set of candidate items cannot be further partitioned into \( k \) partitions. For this reduced candidate set, the algorithm calls a method that can obtain the top-k list of small itemsets for large \( k \) and small itemsets. We informally call the problem of obtaining top-k lists from itemsets where \( k \) is a significant portion of the size of the itemset as the ‘Endgame top-k’ problem. We later describe several ways of implementing this method.

**Example** Figure 2.14 shows the execution of the Crowd-Top-k algorithm to obtain a top-5 list (\( k = 5 \)) of an itemset \( I_0 \) with 320 items, using crowdsourced ranking tasks of size 4. In the figure, we tag each item with a number which is its rank in the baseline ranking. The algorithm is unaware of the baseline ranking. For the top-5 items, which the algorithm aims to retrieve, we use larger italic font. The left part of the figure is the reduction phase, and the right part of the figure is the endgame which crawls back to the initial recursive call to obtain the final top-k list. Initially, the algorithm partitions the itemset into 80 partitions of size 4, and the crowd ranks the items of each partition. We observe that item 4 happens to fall into the same partition with item 1 in this random partitioning. The algorithm forms itemset \( I_1 \) from the maxima of the partitions and calls itself on \( I_1 \). Since \( I_1 \) is a big itemset (80 items), the method partitions it into 20 partitions of size 4 and obtains rankings of the items in each partition using the crowd.
Figure 2.14: Example of execution of the Crowd-Top-k algorithm to obtain the top-5 list of an itemset of 320 items using crowdsourced ranking tasks of size 4

Note that item 5 happens to fall into the same partition as item 3. The method forms set $I_2$ from the maxima of the partitions of $I_1$ and calls itself on $I_2$. The $I_2$ is small (20 items), and cannot be partitioned in more than 5 partitions, therefore, the endgame begins. The algorithm obtains the top-5 list of $I_2$ using a method for the endgame top-$k$ problem. In this example, we obtain the top-5 list $T_2$ by exhaustive comparisons of the items in $I_2$. Top-5 ranked list $T_2$ does not contain items 4 and 5, as they do not belong
to $I_2$. The endgame proceeds by popping the recursive stack to obtain the candidate items of the top-5 list of itemset $I_1$. It forms $C_1$ from the partitions of $I_1$ that contain the top-5 items of $I_2$ and includes items whose rank in $I_2$ can still be less than or equal than $k$. It then obtains the top-5 ranked list $T_1$ of $I_1$ performing exhaustive comparisons in $C_1$. The list $T_1$ includes item 5 retrieved from the partition whose maximum is item 3. The endgame proceeds, forms candidate set $C_0$ from partitions of $I_0$ and $T_1$ and obtains the final top-5 list of $I_0$, which includes item 4 obtained from the partition of $I_0$ whose maximum is item 1. During the backtracking, the size of each of the candidate sets $C_i$ is less than $k \cdot s$, where $s$ is the size of the ranking tasks.

Assuming that the size $s$ and $k$ are small compared to the size of the input itemset, we can consider them fixed and prove the following result.

**Theorem 2.6.1.** Algorithm Crowd-Top-k issues a number of ranking tasks that is linear in the size of the input itemset $I$ and has a logarithmic latency, assuming a bound in the size of partition $s$ and the size of the top-$k$ list.

**Proof.** To reason about the algorithm’s complexity, we first need to calculate the number of recursive calls throughout the algorithm’s execution. At each recursive call, the size of the candidate set is reduced by $s$. For $k \cdot s$ less than the current size of the input itemset the recursion continues. Thus, for the number of recursive calls $r$ the following condition holds:

$$\frac{n}{s^r} \leq k \cdot s \Rightarrow n \leq k \cdot s^{r+1} \Rightarrow \frac{n}{k} \leq s^{r+1} \Rightarrow$$

$$\Rightarrow \log\left(\frac{n}{k}\right) \leq (r + 1) \log s \Rightarrow \log_s\left(n/k\right) - 1 \leq r$$
Thus, the number of recursive calls is:

\[ r(n, k, s) = \max\{\lceil \log_s \left(\frac{n}{k}\right) \rceil - 1, 0\} \]

Denoting the latency of the endgame method with \( L \) (which can be constant for some endgame methods but is by definition constant if we fix \( k, s \)), the overall latency of the \textit{Crowd-Top-k} algorithm is thus:

\[ l_{\text{rec}}(n, k, s) = (L + 1) \cdot r(n, k, s) + L \]

We denote the number of ranking tasks issued by the endgame method, which is a function of \( k \) and \( s \) to be \( U \). The number of ranking tasks is at most:

\[ \text{tasks}_{\text{rec}}(n) = n \cdot \sum_{i=1}^{r(n, k, s)} \frac{1}{s^i} + (r(n, k, s) + 1) \cdot U \]

\[ \leq \frac{n}{s-1} + (r(n, k, s) + 1) \cdot U \]

because the geometric series sum \( \sum_{i=1}^{r(n, k, s)} \frac{1}{s^i} \) is bounded by limit: \( \frac{1}{s-1} \).

Since \( r(n, k, s) \) is logarithmic tasks_{\text{rec}}(n) \in O(n) \) for fixed \( s, k \).

For small top-\( k \) lists (\( k < s \)), where the tournament algorithm is applicable, the \textit{Crowd-Top-k} algorithm would require \( \frac{n}{s-1} + k^2 \cdot \lceil \log_s \left(\frac{n}{k}\right) \rceil \) tasks.

For example, if \( n = 10,000, k = 5 \) and \( s = 10 \), the tournament algorithm would need 2,000 ranking tasks and 11 roundtrips, while the \textit{Crowd-Top-k} algorithm would require roughly 1,200 ranking tasks in 6 roundtrips. The algorithm in [20], which is the same as the algorithm in [11] when we risk no loss for the items in top-\( k \) list, has a latency in \( \Omega(n) \), a fact that can make these methods prohibitively time consuming for
large itemsets. In the presence of randomization, the latency in [11] best-case increases linearly with the size of the top-\( k \) list.

### 2.6.1 Endgame top-\( k \) algorithms

The recursive algorithm Crowd-\( k \) is generic with respect to the method that it uses to obtain the top-\( k \) list when the endgame begins. We are free to choose the algorithm that obtains the top-\( k \) at the endgame, i.e. algorithm endgameTopk of line 3 and line 15 of the pseudocode. Each one has a different tradeoff among cost, latency and quality of answers. We explored several methods for the implementation of the endgame. The four endgame methods described below are applicable to any size of top-\( k \) lists. For smaller top-\( k \) lists, that is, for \( k < s \), the tournament algorithm in also applicable in the endgame.

#### 2.6.1.1 Human-powered sort algorithm

One option is to use the human-powered sorts algorithm [39] which issues a quadratic number of ranking tasks in batches. We implemented the Compare operator proposed in [39] and used it to obtain top-k lists for small itemsets. The number of HITs cannot be smaller than the lower bound \( \frac{n(n-1)}{s(s-1)} \) where \( n \) is the number of items and \( s \) the size of the batch. The algorithm’s advantage is that it issues all necessary tasks in a single roundtrip. For \( n = k \cdot s \), which is the base case of the recursion, it issues a number of comparison tasks in the order of \( O(k^2 \cdot s^2) \).
2.6.1.2 Unbalanced rank estimation

The study in [62] presents a randomized algorithm for obtaining the sorted list of a large itemset by pairwise comparisons. The method is simple; every pairwise comparison has a fixed probability of being chosen and sent to the crowd in a single roundtrip. The items are sorted based on the number of items against which they have won in comparisons. That is, each item’s estimated rank is proportional to the fraction of the items winning it over all items with which it has been compared to. The authors provide an analysis for the expected quality of the results. The quality increases as the sampling rate increases. They prove that the expected accuracy is significantly higher for the top items of the list.

2.6.1.3 Comparisons inference algorithm

We consider a method that is based on inferences due to transitivity. Initially, we randomly pick some comparisons that we issue to the crowd. Then, we choose the comparisons that are most informative, that is, the ones that lead to the highest number of inferred comparisons. The problem of selecting the most informative comparisons bears some similarity to the ‘Next Votes Problem’ that Guo et al. studied in [27]. Finding the optimal solution is NP-hard. The analysis in [27] does not extend to the top-\(k\) problem. Since the optimal solution is difficult, in our study, we employ a heuristic that examines the inference gains for each of the pairwise comparisons and sorts all unknown comparisons by their inference gains.
2.6.1.4 Quick-sort top-\(k\) variant

We consider a variation of quick-sort adapted for obtaining the top-\(k\). The method randomly picks a pivot and finds its rank in the itemset by comparing against all items. The comparison tasks to compute the rank of the pivot are all issued in parallel in a single roundtrip maintaining low latency. The rank of the pivot is the number of items against which it has lost, augmented by 1. If the rank of the pivot is \(k\) the top-\(k\) list is the pivot and all the items that have won the pivot in comparisons. If the rank of the item is larger than \(k\), we can exclude the pivot from the top-\(k\) list along with all the items that it has won. We then repeat the same process and obtain the top-\(k\) list of the items that are still candidate members of the top-\(k\) list.

2.7 Handling inaccuracy of crowds

The key for obtaining high quality output with the Crowd-Top-\(k\) algorithm is to obtain high quality results using crowdsourcing for each of the subsets \(S_i\) of the input itemset \(I\). (line 5 of pseudocode in Figure 2.13)

We assign each ranking task to multiple workers and obtain a single ranking out of the potentially different rankings that workers provide for a particular subset. We use the median rank aggregation in the same way as we do for the tournament algorithm (see Section 2.3.2)
2.7.1 Distributing budget to ranking tasks

Human workers that contribute to Human Intelligence Tasks can be honest or spammers. Spammers can be random spammers or adversarial spammers (also called vandals). For ranking tasks, a random spammer answers with a random ranking of the items of the task, whereas an adversarial spammer provides the correct ranking but in reverse order.

We use two different methods of allocating budget to ranking tasks, the first for the case when we expect random errors and spammers, and a second for when there also exist adversarial spammers in the crowd. In the former case, we use the adaptive technique proposed for the tournament algorithm in Section 2.3.3 that can estimate the difficulty of tasks or the presence of an usually high number of random spammers in a median rank aggregation on the fly, based on the diversity of answers. This technique is directly applicable to the rankings of the Crowd-Top-k algorithm and adaptively uses existing budget by allocating fewer workers to seemingly easy tasks and placing more effort on seemingly difficult tasks. The method is not relevant in the presence of adversarial spammers, because vandals provide the same answer so we cannot use diversity as a red flag. For this case, we use a budgeting strategy based on an analytic estimation of the impact of vandals on the output of the algorithm.

In crowds that undergo quality control (e.g. crowds tested with the Gold Standard Data method of Crowdflower) we generally see zero or negligible percentage of vandals and we only expect random errors and incidental random spamming, as in
fact even honest workers may approximate the behavior of random spammers if fatigue
or other factors lowers their quality. A less moderated crowd such as that of Amazon
Mechanical Turk can have a non-negligible percentage of vandals. As noted in a previous
study [61], the distribution of errors by honest workers is very difficult to obtain, because
it depends on the similarity of items in the baseline ranking. However, it is feasible to
estimate the percentage of vandals in the crowd by sampling from the crowd using a
known dataset. This is because vandals provide a very special answer to ranking tasks;
if the answer we obtain is close to an inverted ranking, we can infer that this is likely
the result of vandalism.

2.7.1.1 Addressing vandalism

In the presence of adversarial spammers, we need to assign the largest possible
number of workers to each aggregation. The suitable number of allocated workers
depends on the budget constraints and the impact that the prevalence of vandals in
a particular aggregation can have on the final output result. We provide an analytic
estimate of this impact based on a simplified error model. Exploiting this analysis,
we devise a principled method for allocating workers, given a particular budget \(Q\) for
ranking tasks. In the experimental study, we check the robustness of the method under
realistic error assumptions.

We assume the percentage of vandals in the crowd is known. In practice, it is
feasible to estimate the actual percentage, by sampling answers from the crowd using a
set of items for which the correct ranking is known, before executing the algorithm.
We approximate reality by assuming that workers are either correct or adversarial.

We define error in the output top-$k$ list, as the percentage of items of the top-$k$ list that we lose; in other words, the number of non-members of the top-$k$ list that appear in the final set of $k$ items that is returned by the algorithm.

We denote with $m_i$ the number of workers that we assign to a ranking task at recursive depth $i$, where $m_0$ is the number of workers we allocate per partition at the initial call of the algorithm. We denote with $z_i$ the number of partitions at recursive call $i$, with $z_0$ being the number of partitions at the initial execution.

We seek the vector $(m_0, \ldots, m_{r(n,k,s)})$ that minimizes the expected error due to vandalism, where $r(n,k,s)$ is the number of recursive calls of the algorithm (see proof of Th. 2.6.1).

We denote the probability that a worker is a vandal, which is roughly equal to the percentage of vandals in the crowd, by $v$. We assume that we have a limited budget $Q$ of crowdsourced ranking tasks.

Picking a worker from the crowd is a Bernoulli trial, so the probability that an aggregation of $m$ workers contains $d$ vandals is $B(m,d) = \binom{m}{d} v^d (1-v)^{m-d}$. When a ranking task is performed by multiple workers and the majority of those workers are adversarial, the aggregation will return an inverted ranking because the median ranks of all items will be inverted. Therefore, the probability that median rank aggregation
outputs an inverted ranking is:

\[ g(m) = \sum_{i=\lceil \frac{m}{2} \rceil}^{m} B(m, i) \]

In the worst case, all the items of the top-\( k \) list that belong to a partition can be lost due to the adversarial behavior. This is because the minimum item of the partition, which is erroneously indicated as maximum, is likely smaller than the maxima of other partitions and would be excluded from the candidate set, along with all the items in the partition.

Since the partitioning is random, the expected number of items of the top-\( k \) list in a partition is \( k/z_i \).

Thus, for the expected error \( e \) due to vandalism of a single partition at recursive depth \( i \):

\[ E(e) \leq g(m_i) \cdot \frac{k}{z_i} \]

Using the union bound of the expected error across all partitions across all recursive calls:

\[ E(err) \leq \sum_{i=0}^{r(n,k,s)} z_i \cdot g(m_i) \cdot \frac{k}{z_i} = k \cdot \sum_{i=0}^{r(n,k,s)} g(m_i) \]

where \( r(n, k, s) \) is the number of recursive calls.

We seek solution \((m_0, ... m_{r(n,k,s)})\) that minimizes the above error bound subject to the budget constraint:

\[ \sum_{i=1}^{r(n,k,s)} z_i \cdot m_i \leq Q \]
This is an optimization problem that resembles a non-linear variant of the knapsack problem (assigning a negative sign to the error bound to convert it to a maximization problem). Solving the above optimization problem provides us with a budgeting strategy that distributes the available budget to ranking tasks across recursive calls. For large itemsets that require several recursive calls, exhaustive search of all combinations that maximally satisfy the budget constraint is infeasible due to combinatorial explosion. We thus employ a greedy approach similar to the ones used for knapsack to obtain an approximately optimal solution to the optimization problem.

2.8 Randomized variant

We propose and implement a randomized variant of the recursive algorithm Crowd-Top-k that reduces the cost of the endgame by undertaking a small risk of losing items of the top-k list from the final result.

The key observation that allows us to devise a randomized algorithm is that forming the new candidate set using the top-k maxima and items from partitions where the top-k maxima belong is necessary because of the non-zero probability that more than one items of the top-k list of the given itemset may fall in the same partition. Therefore, the top-k list $\hat{T}_m$ of the maxima, and the top-k list of the itemset always have a non-empty intersection but are often not the same.

We call the event that exactly $w$ items of the top-k list of the itemset fall in the same partition a $w$-fold collision. The top-$w$ item of a partition whose maximum
belongs to $\hat{T}_m$, can belong to the top-$k$ list of $I$ with a probability that is at most the sum of the probabilities of $f$-fold collisions taking place, for all $f$ where $w \leq f \leq k$. If this bound is low enough so that it is in practice negligible we can keep in the candidate set only the top-$(w - 1)$ items of the partitions whose maxima belong to $\hat{T}_m$ instead of the number of items that are necessary in the standard version of the algorithm. In the extreme case where a $f$-fold collision, where $f \geq 2$, is highly unlikely, executing the endgame is redundant; the top-$k$ list of the itemset is most likely $\hat{T}_m$ itself, i.e. the top-$k$ list of the maxima of the partitions. We can quantify an upper bound of the probability of a $w$-fold collision happening. As an example, for an itemset of 1,000,000 items, partition size of 5 and $k = 10$, the probability that more than one of the items of the top-$k$ list fall in the same partition cannot exceed $2.5 \cdot 10^{-4}$. Thus, skipping the execution of the endgame and returning the top-$k$ list of the maxima $\hat{T}_m$ as the top-$k$ list of the entire itemset carries a risk of at most 0.025% of missing items of the top-$k$ list. We can deem this as low, and in many cases, it is significantly lower than the risk of losing items due to vandalism or errors by the crowd.

Formally, the probability $q(w)$ of a $w$-fold collision occurring at a given partition at recursive call $i$ is:

$$q(w) = \binom{k}{w} \cdot \frac{1}{z_i}^w \cdot (1 - \frac{1}{z_i})^{k-w}$$

and for the probability across all partitions at recursive call $i$ we can use the union bound to obtain: $Pr[w, i] \leq z_i \cdot q(w)$.

The probability $r(c, i)$ of losing an item of the top-$k$ at recursive call $i$ by
promoting only $c \cdot k$ items to the endgame methods instead of $O(s \cdot k)$ is bounded by:

$$\sum_{w=c+1}^{w=k} z_i \cdot q(w).$$

The randomized variant takes a risk threshold as additional input. This represents the level of risk of losing items of the top-$k$ list due to the randomization which the user of the algorithm deems acceptably low. The algorithm then pre-calculates the value $c_i$ for each recursive call $i$, which maintains the total risk bound less or equal to the risk threshold, according to the above upper bound. During the execution of the algorithm, at each recursive call $i$, the algorithm invokes the endgame top-$k$ method for $c_i$ items of each partition, which correspond to the top-$c_i$ items of partitions where items of $\hat{T}_m$ belong, instead of $O(s)$ items in the non-randomized case. For $c_i = 1$, there is no invocation of the endgame top-$k$ algorithm; the algorithm returns the top-$k$ list of the maxima $\hat{T}_m$ as the top-$k$ list of the itemset.

## 2.9 Experimental study with simulated crowds

### 2.9.1 Methodology

For the data, we used the same equi-distance model for the qualities of items, and similarly the human workers who make random mistakes again follow the Thurstonian model. We introduce a new category of spammer workers, adversarial spammers who provide reverted answers as described in Section 2.7.1.1.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Range</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of itemset</td>
<td>1,000-100,000</td>
<td>10,000</td>
</tr>
<tr>
<td>k (size of top-k list)</td>
<td>5-50</td>
<td>50</td>
</tr>
<tr>
<td>Size of partition</td>
<td>-</td>
<td>2-10</td>
</tr>
<tr>
<td>Random spam</td>
<td>10%-40%</td>
<td>20%</td>
</tr>
<tr>
<td>Vandals</td>
<td>1%-15%</td>
<td>1%</td>
</tr>
<tr>
<td>Error rate</td>
<td>10%-40%</td>
<td>25%</td>
</tr>
<tr>
<td>Budget</td>
<td>4K-1.5M</td>
<td>15K</td>
</tr>
</tbody>
</table>

The above table shows how we vary the experimental parameters and their default values.

**Error measure.** The error measure we use is the same as we define in Section 2.2.2.

**Cost metric.** We model the amount of work, and thus the cost, of a HIT as $c = \frac{s \cdot \log_2 s}{2}$, where $s$ is the size of the ranking tasks. Though it is difficult to quantify the mental effort of humans who complete ranking tasks, this cost measure assumes that the effort does not grow quadratically with the size of the ranking task (as a bubble-sort like approach would require), and is reminiscent of the complexity of sorting methods, such as quicksort and mergesort. In some cases, we avoid utilizing the cost measure by using pairwise comparisons (equivalent of a ranking task of size 2) in the reduction phase. If all HITs have the same size throughout the execution of the algorithm, the amount of work is proportional to the number of HITs. Otherwise, HITs with higher sizes require more effort. We also cap the ranking task size to 10 (in fact, we only report results for ranking tasks of size 4 and 10), as in reality workers would be reluctant or unable to complete very large ranking tasks.
2.9.2 Results

**Comparative study for small top-k lists.** We compare the Crowd-Top-k algorithm against the tournament algorithm which is applicable to small top-k lists (smaller than the size of a ranking task). We set the minimum number of workers per aggregation to 3, for high quality. We use the tournament algorithm itself as the endgame method of the recursive algorithm Crowd-Top-k. We also compare against the algorithm in [20] using roughly 100,000 pairwise comparisons, which is equivalent to 6000 ranking tasks of size 10 according to the cost metric.

Figure 2.15 shows the result of an experiment that obtains the top-5 list of 10,000 items with a fixed budget of 6,000, default spammer percentage and increasing error rate. The Crowd-Top-k algorithm performs better for all error rate levels while the spread between the performance of the two methods increases for higher error rates. For an error rate of 10% the Crowd-Top-k algorithm yields results with almost zero error. The results confirm the superiority of the recursive approach to the tournament algorithm. The number of roundtrips fluctuates for the two methods, ranging from 11 to 30, due to the adaptive allocation of tasks to human workers. The method of [20] has a very high latency (> 10,000 roundtrips) which makes it unsuitable in real settings and for this setting it also achieved poorer quality results than the recursive algorithm.

**Performance of the Crowd-Top-k algorithm for large top-k lists (k > s).** We conducted experiments with several instantiations of the Crowd-Top-k algorithm for large top-k lists using different methods for the endgame.
The human-powered sort algorithm [39] for the endgame was highly inefficient for the top-$k$ problem in terms of required budget and tolerance to errors, so we omit the results of its performance. We tune the methods we tested appropriately to use the same budget so that the comparison is fair and meaningful. The budget is set to 15,000 pairwise comparison tasks ($s = 2$) for obtaining the top-50 list of an itemset of 1,000 items. We report the results of five different methods to obtain the top-$k$ list:

- Unbalanced rank estimation (URE) proposed in [62] to retrieve the top-$k$ list out of the entire itemset.

- Recursive algorithm Crowd-Top-$k$ with unbalanced rank estimation (URE) as the method for the endgame.

- Recursive algorithm Crowd-Top-$k$ with the comparisons inference algorithm as the method for the endgame. We obtain the same number of pairwise comparisons (three times the size of the dataset) from the crowd in all roundtrips.
• Recursive algorithm Crowd-Top-k using the quick-sort top-k variant algorithm for the endgame.

• Method described in [20] for the top-k problem which has the same order of latency with [11] and it is in fact the same algorithm when it is not randomized.

Figure 2.16 shows the results of the five methods for default error noise and increasing percentage of random spammers. The unbalanced rank estimation method has the smallest latency as it uses a single roundtrip but provides the poorest quality results. The Crowd-Top-k algorithm using the quick-sort top-k variant in the endgame is the method that makes the most efficient use of the budget providing the highest quality output. The method of [20] is providing results of comparative quality, yet, the latency is prohibitively high (>2000 roundtrips). The Crowd-Top-k method using unbalanced rank estimation for the endgame maintains the error low and the latency at 9 roundtrips. It is a balanced approach that provides low output error with low latency. The Crowd-Top-k method with the comparisons inference algorithm provides good results comparable to those of the quick-sort top-k variant for lower spammer percentages and with lower latency but as the spammer percentage increases the output error approaches that of the unbalanced rank estimation. The comparisons inference algorithm is vulnerable to error propagation which explains the increase in the output error as spammers increase.

**Handling of vandals.** We described a method that returns the number of workers that we should allocate to the ranking tasks at each recursive call for a given
Figure 2.16: Comparative performance of methods for large top-$k$ lists

budget, in the presence of vandals (adversarial spammers). The probabilities we use in our analysis were upper bounds obtained from union bounds and may not be tight.

A question we need to address is whether the budgeting strategy works efficiently in practice. We conducted experiments to evaluate the performance of the budgeting strategy. For an itemset of 1,000 items, $s = 2$ and $k = 20$ the algorithm is called 5 times, one initially and 4 recursively. The budgeting algorithm returns a vector $V = (m_0, ... m_4)$ where $m_i$ is the number of workers per partition at recursive call $i$, with $m_0$ being the workers per partition at the initial call of the algorithm. If the budgeting strategy provides a roughly optimal budget arrangement we expect the error of the output to increase when we use a different budget arrangement. In particular, we expect the error to get higher as the distance of the vector that represents the budget arrangement from the vector of the optimal budget arrangement $V$ increases.

Figure 2.17 demonstrates the error for three levels of vandals in the crowd,
Figure 2.17: Performance of budgeting strategy

at 5% and 10% and 15%. We evaluate the Crowd-Top-

k algorithm with URE for the endgame using several budget arrangements that maximally satisfy the budget constraints, that is, even one pair of extra workers at some stage would exceed total available budget for ranking tasks which stands at 5,000. The total budget stands at 17,000. The x-axis represents the Manhattan distance to the vector $V$ that the budgeting method returns as optimal arrangement.

We assume that the estimation of the vandal percentage of the crowd stands at 10%. The random errors performed by honest workers are at the default level. For an actual percentage of vandals at 10% we observe that in the vicinity of vector $V$, that the budgeting strategy returns, we obtain the lowest error results. It is reasonable to assume that the vandal percentage estimate may be an underestimate (e.g. -50%) or an overestimate (e.g. +50%), since it is the result of sampling. We thus report results for the same level of vandal percentage estimate at 10% but for the actual levels of budget
standing at 5% and 15%. We observe that the results are qualitatively the same as when
the actual percentage is at 10%, and the lowest error is in the vicinity of the vector $V$
of the budgeting strategy.

The results confirm the relevance of our analysis and show that our strategy
distributes the budget efficiently across the algorithm’s stages even for difficult compar-
isons where honest workers do not always provide correct answers. In particular, for
budgeting vectors that are close to $V$ the performance is virtually unchanged (can be
slightly inferior, or even slightly superior as $V$ is not necessarily the real optimal ar-
rangement since it is derived from an analysis based on union bounds), while for vectors
with high distance from $V$, the error increases in all cases. We also observe a small
sensitivity of the budgeting strategy to errors in the estimate of the vandals percentage
from crowd sampling.

2.9.2.1 Performance of randomized variant

We expect the randomized approach to be more efficient for very large itemsets
and large $k$’s. This is because for large $k$’s we feed many items to the endgame, and the
number of pairwise comparisons increases quadratically with $k$. Also, for large itemsets
the risk of $w$-fold collisions during partitioning decreases. We evaluate the randomized
recursive algorithm Crowd-Top-$k$ with URE for the endgame, for a large itemset of
100,000 items, $k = 50$, and partition size of 10. The size of comparison tasks is different
in the reduction stage of the algorithm and the endgame stage (10 and 2 respectively).
In the interest of fairness, we report the amount of work according to the cost metric
rather than the number of HITs. Otherwise, reporting only the number of HITs would favor the results, since the randomized approach achieves budget saves only for the comparison tasks that take place in the endgame stage, which in this case are smaller and therefore require less effort.

Figure 2.18 demonstrates the error as the risk increases from zero where there is no randomization to $3 \cdot 10^{-4}$ for default errors by honest workers and vandals at 1%, 3% and 5%. The risk is the probability that we lose some item of the top-$k$ list from the final result, due to the randomization. We observe that the error is essentially unchanged and even paradoxically decreases slightly instead of increasing in some cases. This happens because the risk of loss remains negligibly low, but the number of items that we send to the URE method drops significantly. For constant parameter $c = 200$ of the URE method, the sampling rate increases, and for a smaller dataset this leads to an enhanced output. Still, the required budget decreases because the number of all possible pairwise comparisons decreases quadratically, and we require significantly less
Figure 2.19: Example of an increased difficulty pairwise comparison task with polygons addressed to real workers of Amazon Mechanical Turk

budget to achieve comparable or even superior quality. We can see the use of budget, expressed in required amount of work, in the following table.

<table>
<thead>
<tr>
<th>Risk</th>
<th>0</th>
<th>$3 \cdot 10^{-8}$</th>
<th>$3 \cdot 10^{-6}$</th>
<th>$3 \cdot 10^{-4}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amount of work (in million)</td>
<td>1.511</td>
<td>0.908</td>
<td>0.782</td>
<td>0.649</td>
</tr>
<tr>
<td>Budget save</td>
<td>0%</td>
<td>42%</td>
<td>48%</td>
<td>57%</td>
</tr>
</tbody>
</table>

The figures in the table above show that we achieve a budget save that approaches 50% for a very low risk of $3 \cdot 10^{-6}$ and even larger saves for higher levels of risk, without noticeable change in the quality of results.

2.10 Experimental study with real crowds

2.10.1 Methodology

Data. We tested three types of item sets of cardinality 320. For $k = 5$ and $s = 4$, the algorithm runs in the same way as the example of figure 2.14, yet, we used quick-sort style endgame instead of exhaustive. The first item set is a set of squares whose edges range from 60 to 380, increasing one pixel at a time. The difference in
size among squares is easy to discern, therefore, we consider the overall task of finding the top-5 squares as an easy task. Similarly to the experiment for the tournament algorithm, the second itemset, which we deem of medium difficulty, is a set of polygons with vertices 4-10 alternately and their area increasing by a constant factor, a unit area, while for the hard task we decrease the size of the unit area so that the polygons become very difficult to compare. In figure 2.19 we can see a snapshot of a pairwise comparison task among polygons.

**Workers.** We used the workers of Amazon’s Mechanical Turk without imposing any restrictions nor requiring qualification tests. We paid $0.01 per task, which is the lowest rate for a task. The application shuffles the shapes at each round as dictated by the algorithm at every roundtrip, constructs HTML forms with the shapes on the fly, and uploads the forms to a certified server of UC Santa Cruz through SFTP. Uploading to a certified server was necessary to ensure that workers can fetch the data through HTTPS and submit the results without security warning messages from their browser. If a HIT remained unanswered for more than 25 minutes, it was more likely that it would remain so indefinitely, since new tasks would come on the top of the list of tasks at Mechanical Turk’s page making our tasks less visible. Due to this, the application automatically reposts belated HITs after that time, disposing the previous ones.

**Error measure.** The error measure is the same as for the simulations, having the areas of the squares as values, and considering the area difference among consecutive shapes in the baseline ranking to be the unit area for all three datasets.

**Algorithms.** We tested the *Crowd-Top-k* algorithm with quick-sort style
endgame. We allowed for a risk of loss of less than 2% by allowing at most 3 items from each partition to proceed to the endgame. First round uses 1 worker per item and second round 3. Using 7 workers for each comparison in the endgame this corresponds to roughly 800 HITs overall, of which 160 were ranking tasks and the rest pairwise comparisons. This corresponds to a cost of approximately 1220 according to our cost metric, though for both tasks we paid the same lowest possible rate and the workers contributed to them with no problem. It is fair though to reason in terms of cost since a hypothetic platform may allow for lower compensation rates. We also tested the algorithm in [11] assuming $\log X = 1$ which roughly corresponds to $\delta$ around 15%. This requires roughly 800 pairwise HITs when posting 3 comparisons in the upper levels of the $X$-tree. Finally, we tested its non-randomized version which is essentially the algorithm in [20] requiring roughly 1,300 pairwise comparison HITs.
2.10.2 Results

The results in figure 2.20 demonstrate a comparable performance of the three tested methods in terms of quality of results for all three levels of difficulty. What is significantly different is the difference in latencies. The Crowd-Top-k has an approximate latency of 13 roundtrips and 72 minutes, while the algorithm of [11] has a latency of 214 roundtrips and 508 minutes (there is only one task in each roundtrip but the total time is significantly higher than the Crowd-Top-k’s latency, because of the parallel execution of tasks in each roundtrip of Crowd-Top-k). The non-randomized version has a latency of 420 roundtrips and roughly 1170 minutes. The latency of those methods would get prohibitively high for even bigger datasets as it would increase linearly, while the Crowd-Top-k algorithm would scale as the latency increases logarithmically.

2.11 Conclusions

The techniques we propose make efficient use of available budget and overcome major limitations of prior art. The algorithms have the flexibility to use different methods for solving the endgame problem based on the preferred tradeoffs for latency and quality of results, given a specific budget. They demonstrate high tolerance to random spammers, vandals and errors even for unrealistically high spammer percentages and errors. Applying the randomized approach to very large itemsets, we can reduce the necessary cost drastically, with negligible risk of lowering the quality of results.
Chapter 3

Reliable Aggregation of Boolean Crowdsourced Tasks

3.1 Problem setting and contributions

In this chapter, we focus on the unsupervised binary crowdsourcing problem, where workers answer Yes/No questions about items. An example of a binary crowdsourcing problem is to determine whether a Wikipedia edit is vandalism, or whether a given webpage is appropriate for children. In the presence of historical data and multiple input from same users on different tasks, it is natural to assume that there are ways to analyze the workers’ activity and derive more accurate answers by isolating systematic spammers or low quality workers. We study and propose methods that can improve results by inferring the reliability of users. One can view the problem as a process of probabilistic inference on a bipartite graph with workers and items as nodes,
where both worker reputations and item labels are unknown. The problem of indirect inference of human reputation was first studied in the 70s, long before the advent of the internet and crowdsourcing marketplaces, with the description of the Expectation Maximization algorithm [12]. Approaches closely related to EM were proposed in [52] and [50]. A variational approach to the problem was recently proposed in [34] and a mean field method was proposed in [38]. The approach of [34], which we abbreviate by KOS, is proved to be asymptotically optimal as the number of workers tends to infinity, provided there is an unlimited supply of statistically independent workers. Nevertheless, we will show that KOS is not optimal in extracting the best estimate of the underlying truth from a finite amount of work performed.

We begin by describing a general framework for binary inference, in which a beta-shaped belief function is iteratively updated. We show that the KOS approach corresponds to a particular choice of update for the belief functions. Casting the KOS algorithm as a belief-function update enables us to gain insights on its limitation. In particular, we show that the KOS approach is not optimal whenever the amount of work performed is non-uniform across workers, a very common case in practice, as well as whenever there is correlation between the answers provided by different workers. Furthermore, in cases involving a finite number of workers and items, correlation is created simply by iterating the inference step too many times; indeed, we show that the performance of KOS generally gets worse with increasing number of inference iterations.

We describe two variations of the beta-shaped belief function update, which we call the harmonic and the parameter estimation algorithms [14]. The harmonic update
is a simple update equation that aims at limiting the effects on the final result of worker correlation, worker supply finiteness, and difference in amount of work performed by different workers. There is no deep theoretical underpinning for the harmonic approach, and its virtues are its simplicity and empirical robustness. The parameter estimation approach, in contrast, consists in updating the belief beta-distributions by estimating, at each iteration, the beta-distribution parameters that provide the best approximation for the true posterior belief distribution after one step of inference. We develop in detail the parameter estimation procedure, showing that it is feasible even for large practical crowdsourcing problems.

For the purpose of this study, we model the user reputations by a one-coin parameter, where there is only one distribution representing each user’s reliability. If the distribution of a user has, for example, mean 0.75, this means that the user will report the correct answer with probability 0.75. But there can be a more complex model describing the user’s behavior, for instance, a two-coin model where the user has a different accuracy for the positive labels and a different accuracy for the negative labels (represented by two separate user distributions). The ideas we describe for performing the update user and item distributions are extensible to a more complex two-coined model, such as the two-coin extension of [12] and [38], where it is assumed that users can have different true positive and true negative rates. While our empirical study focuses on unsupervised settings, supervised variants with our methods are possible, as the methods maintain distributions on both user reputations and item qualities, and we can use knowledge on the crowd or the items to impose informative priors on the
distributions.

We evaluate the harmonic and parameter estimation approaches both on synthetic data, and on large real-world examples. On synthetic data, we show that for non-regular graphs and for correlated responses, both our approaches perform well, providing superior performance compared with the KOS and EM methods. We then consider four real-world datasets generated by other research groups via Amazon Mechanical Turk. One dataset, kindly provided by the author of [49], consists of Wikipedia edits classified by workers according to whether they are vandalism; other two datasets contain annotations by non-experts on questions of textual entailment and temporal ordering of natural language texts [53], and a fourth dataset comes from the Duchenne experiment [65]. The parameter estimation approach shows statistically significant superiority against other methods with respect to average recall for two of the real-word datasets, while it ties with other methods in the other two cases. The harmonic approach provides performance closely approaching that of parameter estimation, while maintaining the advantage of simplicity and ease of implementation.

Overall, the experiments show that the harmonic and parameter estimation approaches provide robust approaches to binary crowdsourcing that improve on the state of the art in a wide variety of settings.
3.2 Notation and previous work

3.2.1 Notation

We consider a set $U$ of users and $I$ of items. A task consists in a user $u \in U$ giving an answer $a_{ui} \in \{+1, -1\}$ on item $i \in I$; by convention, $+1$ denotes a positive answer (i.e. ‘true’, or ‘yes’) and $-1$ a negative answer. We denote the answer set by $A$. The set of users, items and answers form a bipartite graph $E \subseteq U \times I$, whose edges represent the users’ votes on the items. We call the users that have voted on an item $i \in I$ the neighborhood of $i$, denoted by $\partial i = \{u \mid (u, i) \in E\}$. Likewise, the neighborhood $\partial u$ of a user $u$ consists of the set $\{i \in I \mid (u, i) \in E\}$ of items that $u$ voted on. The goal of the binary crowdsourcing problem consists in aggregating the user votes into item labels. One can view this as a double inference task: infer the most likely reliability of workers (which we can view as latent variables) based on their answers, and then use the worker’s inferred reliabilities to infer the most likely labels of the items.

3.2.2 Previous work

Viewing this as problem of probabilistic inference on a graphical model (the bipartite graph), its optimal solution is intractable [37]. There exist several approaches to tackle this problem [34, 12, 38, 50].

A recent approach to the binary crowdsourcing problem is described in [34]. The approach is closely related to belief propagation (BP) [67], and executes on the
bipartite voting graph in the style of BP by passing messages from workers to items and back in iterations. We give the pseudocode of KOS approach in Figure 3.1. The algorithm maintains messages of users addressed to items (quantities $y_{u \rightarrow i}$). We can view each message as a single value that represents the reliability of a user. The higher the value the higher the user’s reputation. Negative values represent negative reputation. The reliabilities of users’ are initialized randomly by sampling from Gaussian distributions with mean 1 and variance of 1 (in this manner, the majority of users have positive initial reputation). The algorithm also maintains messages of items addressed to users (quantities $x_{i \rightarrow u}$), which can be viewed as the current perceived quality of the item, with positive values representing a high quality and negative values a negative quality. The votes of users on items, together with the messages of users addressed to items, are used to update the messages of items to users in an additive manner. Negative votes (i.e. $a_{iu} < 0$ for a pair $(i, u)$) count negative, that is, reduce the value of $x$ by the amount of the reputation $y$ of the user who voted against it, and likewise positive votes by reputed users increase the value of $x$ of the item message. Likewise, a negative vote by a user with negative reputation, will increase the perceived quality of the item.

Once the item messages $x$ are updated in this manner, they are in turn used to update the users’ messages $y$, penalizing the reputation of users who have voted against good quality items (or in favor of bad quality items) and rewarding the reputation of users who voted in favor of good quality items (or against bad items). Since for each user we hold a message to an item (so for each user we hold a number of messages equal to the number of items in the user’s neighborhood), we exclude the self-votes, that is,
Input: bipartite graph $E$, answers $a_{ui}, k_{\text{max}}$

Output: Estimation of correct solutions $s_i \in \{+1, -1\}$ for all $i \in I$.

1 foreach $(u, i) \in E$ do
2  Initialize $y_{u \rightarrow i}$ with $Z_{i,j} \sim N(1,1)$;
3 for $k = 1, \ldots, k_{\text{max}}$ do
4  foreach $(u, i) \in E$ do
5     $x_{i \rightarrow u}^{(k)} \leftarrow \sum_{u' \in \partial_i \setminus u} a_{iu'} \cdot y_{u' \rightarrow i}^{(k-1)}$;
6  foreach $(u, i) \in E$ do
7     $y_{u \rightarrow i}^{(k)} \leftarrow \sum_{i' \in \partial u \setminus i} a_{i'u} \cdot x_{i' \rightarrow u}^{(k)}$;
8 foreach item $i$ do
9     $x_i \leftarrow \sum_{u' \in \partial i} a_{iu'} \cdot y_{u' \rightarrow i}^{(k_{\text{max}}-1)}$;
10 Return estimate $\hat{s}_i = [\text{sgn}(x_i)]$ for all $i \in I$.

Figure 3.1: KOS algorithm for labeling items using binary crowdsourced answers.

The perception of the item’s quality by a given user does not take into account the user’s vote on the item. The iteration goes back and forth for a specific number of iterations. After the iterations are over, the estimated label of each item (+1 or -1) is obtained from the sign of item messages $x$. The number of iterations $k_{\text{max}}$ can be arbitrarily large but in practice we need only a few iterations until there is no change in the signs of messages $x$. The value of the internal messages will keep increasing (or decreasing in the case of bad quality items) with the extra iterations, but after the initial oscillations...
in the sign of the messages, there are no sign flips. For a formal proof of the algorithm’s properties and convergence we refer to the technical results in [36].

The authors present results on synthetic graphs showing the superiority of this method to EM [12], and prove that the approach is asymptotically optimal for regular graphs, that is, as the size of the graph tends to infinity their approach is up to a constant factor as good as an oracle that knows the reliability of the workers.

Liu et al.[38] propose a variation to the EM method [12]. Making a beta distribution assumption for the probability \( q_j \) that a user \( j \) will provide a correct answer, the M-step update for their method is obtained using a variant written in terms of the posterior mean of the beta distribution rather than its posterior mode. The authors argue that this variation plays a role of Laplace smoothing. They report results that display a superior performance compared with EM and comparable performance to KOS [34] for some informative priors on users’ quality. They also explore different models for users’ voting. Instead of assuming a fixed reliability of users, they examine a two-coin model where each user has a varying reliability based on the task (specificity). Alternative models for user behavior have been considered and appear applicable in tasks requiring expertise such as the Bluebird Dataset [63].

3.2.3 Beta belief distributions for users and items

We will show below that the KOS algorithm, and our algorithms, can be interpreted as update rules for beta distributions of belief on item value and user quality. This will provide an unifying framework to understand the properties of KOS and of
our algorithms. Note that our setting of beta belief updates is not a variant of belief propagation; despite of the use of the term ‘belief’ in both cases, we maintain real-valued distributions of users reputations and item qualities, unlike belief propagation.

Similarly to [38], we characterize users and items with probability distributions in the domain $[0, 1]$. The distribution of a worker represents the information on the worker’s reputation or reliability, while the distribution of an item represents the information on its quality. The smaller the standard deviation of the distribution, the "peakier" the distribution, and the smaller the uncertainty over item quality or worker reliability. The higher the mean of the distributions, and higher the expected quality of workers or the expected truth value of items. A worker of perfect reliability has distribution $p(r) = \delta(r - 1)$ and a perfectly unreliable worker has distribution $u(r) = \delta(r)$, where $\delta$ is the Dirac delta function. A natural choice for the distributions over worker reliability and item quality is the beta distribution. A beta distribution $\text{Beta}(\alpha, \beta)$ with parameters $\alpha, \beta$ represents the posterior probability distribution over the bias $x$ of a coin of which we saw $\alpha - 1$ heads (positive votes for items, truthful acts for users) and $\beta - 1$ tails (negative votes for items, false acts for users), starting from the uniform prior. An item whose distribution has $\alpha > \beta$ will have distribution median greater than 0.5, and be classified as true at the end of the inference process; conversely if $\alpha < \beta$.

The presentation of the KOS algorithm is slightly complicated by the fact that the algorithm, when computing the feedback to item $i$ from users in $\partial i$, avoids considering the effect of $i$ itself on those users. This leads to the message-passing presentation of the method that we see in Figure 3.1. If we allow the consideration of
the effect of $i$ on $\partial i$, we obtain a simpler version of KOS that “allows self-influence”.

Such a version can be described succinctly as follows. For every user $u$, initialize its reputation via $r_u \sim N(1,1)$. Then, iteratively perform the updates:

$$r_i = \sum_{u \in \partial i} r_u a_{ui} \quad r_u = \sum_{i \in \partial u} r_i a_{ui}.$$  \hspace{1cm} (3.1)

Note that, at a step, the influence of user $u$ on item $i$ is that the amount $r_u a_{ui}$ is added to $r_i$ (and similarly in the other direction, from items to users). After the desired number of iterations, decide the value of $i$ by $\hat{s}_i = \text{sign}(r_i)$, for all $i \in I$.

We can view this algorithm as an update rule for beta distributions as follows. Every user $u$ is associated with a beta distribution $\text{Beta}(\alpha_u, \beta_u)$ representing their truthfulness, and every item $i$ is associated with the distribution $\text{Beta}(\alpha_i, \beta_i)$ representing its quality. Our interpretation maintains the invariants $r_u = \alpha_u - \beta_u$ and $r_i = \alpha_i - \beta_i$.

Initially, we set $\alpha_u = 1 + r_u, \beta_u = 1$ for every $u \in U$, where $r_u$ is initialized from the normal distribution as before. Since $r_u$ is sampled from the normal distribution with mean and variance 1, it is more likely than not a positive number. Similarly to KOS method’s user messages $y_{u \to i}$, this initialization will make the algorithm consider the majority of users slightly reliable in the beginning, as the mean of the corresponding Beta distribution $\frac{\alpha_u}{\alpha_u + \beta_u}$ of the user will be larger than 0.5.

To perform the update step of (3.1), for each $i \in I$ we initialize $\alpha_i = \beta_i = 1$, and for each $u \in \partial i$, we increment $\alpha_i, \beta_i$ as follows:

$$\begin{align*}
\alpha_i &:= \alpha_i + \alpha_u, \quad \beta_i := \beta_i + \beta_u \quad \text{if } a_{ui} > 0, \\
\alpha_i &:= \alpha_i + \beta_u, \quad \beta_i := \beta_i + \alpha_u \quad \text{otherwise}.
\end{align*}$$  \hspace{1cm} (3.2)

80
A similar update is then performed for each $u \in U$. We can prove by induction that the above beta-distribution based algorithm, and the simplified (3.1) algorithms, are equivalent. To reason accurately, and to take into account that the KOS method maintains a item message for each user $y_{u \rightarrow i}$ for all item’s in the user’s neighborhood, and likewise a worker message for each item $x_{i \rightarrow u}$ for all workers in the item’s neighborhood, where the self-votes are not taken into account, we assemble a frame of probability distributions that allows us to interpret the KOS approach as a method of likelihood updates for users’ reputations and items’ qualities.

### 3.2.4 Iterative updates of user and items distribution

Using the distributions for workers and items, we can create a framework of iterative updates. We start with an agnostic view of items quality distributions $f_i(x)$, assigning to the shape parameters of the beta distributions $\alpha = \beta = 1$ which yield the uniform distribution. Workers are given either random distribution shape parameters (in the same manner as KOS algorithm [34]), or a slight edge of parameter $\alpha$ over $\beta$ that represents a slight initial belief in their truthfulness.

We use $\hat{\alpha}_u$ and $\hat{\beta}_u$ to denote the shape parameters of workers’ beta distributions. The worker’s distributions are thus of the following form:

$$g_u(r) = r^{\hat{\alpha}_u} \cdot (1 - r)^{\hat{\beta}_u}$$

We will describe a particular form of iterative update. After giving its description, we will prove that it is equivalent to the KOS method and describe its deficiencies,
which are also the deficiencies of the KOS approach. Then, we will describe a different method of updates. We update the items’ distributions \( f_i(x) \) by using the reputation distributions of workers together with their binary answer as a likelihood function in the following manner: if a worker votes in favor of an item, the likelihood function is the worker distribution itself, otherwise the likelihood function is the reflection of the worker distribution about the 1/2-axis. We make an independence assumption for the answers that the workers give and thus we can update the item quality distributions as follows:

\[
 f_i^{(k+1)}(x) = \prod_{u \in \partial_i} g_i^k(w_{iu}x + (1 - w_{iu})(1 - x))
\]

Then, the worker’s reputations are updated using the items qualities as likelihood functions. Likewise, if a worker has voted in favor of an item, the likelihood function is the worker’s distribution itself, otherwise, it is the reflection of the distribution about the 1/2-axis. This way, the reputation of a worker who has voted in favor of a high quality item will be updated in the worker’s favor. The opposite will occur for a worker who has voted in favor of a low quality item.

The worker’s reputation update is thus the following:

\[
 g_u^{(k+1)}(r) = \prod_{i \in \partial_u} f_i^{(k+1)}(w_{iu}r + (1 - w_{iu})(1 - r))
\]

### 3.2.5 A beta-distribution interpretation of KOS

We consider a variation where we do not hold a single reputation distribution per worker, but a per item reputation distribution. Likewise, for item qualities, we hold
a separate item quality distribution for each worker than has voted on the item. In this way, we end up with a large number of distributions, which is $O(mn)$, where $m$ is the number of items and $n$ the number of users. Initializing the worker’s reputation as before, the updates of the distributions are given by the formulas below, effectively excluding the self-votes from the updates:

$$f^{(k+1)}_{iu}(x) = \prod_{u' \in \partial \setminus u} g^{k}_{iu'}(w_{iu'}x + (1 - w_{iu'})(1 - x))$$

$$g^{(k+1)}_{ui}(r) = \prod_{i' \in \partial \setminus i} f^{(k+1)}_{ii'}(w_{ii'}r + (1 - w_{ii'})(1 - r))$$

**Theorem 3.2.1.** The method of section 3.2.5 using beta distributions for the initialization of worker’s reputations is equivalent to the KOS algorithm

**Proof.** We aim to prove that after each iteration of the algorithm the difference of the shape parameters $\alpha - \beta$ of the workers distributions is equal to worker messages of KOS algorithm, and the difference of the shape parameters of items’ qualities is equal to the item messages of the KOS algorithm. Thus, the messages of the KOS algorithm and consequently its final output is fully recoverable by executing the iterative updates of 3.2.5. We will prove this claim by induction. Initially, worker’s reputations are assigned random shape parameters. Workers answers, except for workers $u$, for a specific item $i$ will update distribution $f_{iu}$ of item quality $i$ as perceived by worker $u$ (i.e. as viewed by all workers weighted by their reputation, excluding his vote). For a total of $q$ workers that voted on item $i$, we tag with numbers $1...p$ the workers who voted favorably on $i$, and $p + 1...q$ the workers who voted negatively. Due to addition of exponents by multiplication and the fact that we use the reflection of the workers
reputation distribution as the likelihood function when workers have voted against an item, the shape parameters of \( f_{iu} \) are:

\[
\alpha_{iu} = \sum_{j=1}^{p} \hat{\alpha}_{ij} + \sum_{j=p+1}^{q} \hat{\beta}_{iu} 
\]

and

\[
\beta_{iu} = \sum_{j=1}^{p} \hat{\beta}_{ij} + \sum_{j=p+1}^{q} \hat{\alpha}_{iu} 
\]

Thus:

\[
\alpha_{iu} - \beta_{iu} = \sum_{j=1}^{p} (\hat{\alpha}_{ij} - \hat{\beta}_{ij}) - \sum_{j=p+1}^{q} \hat{\alpha}_{iu} - \hat{\beta}_{iu} = \sum_{j=1}^{q} (a_{ui} \cdot (\hat{\alpha}_{iu} - \hat{\beta}_{iu}))
\]

Initializing \( y_{i \rightarrow u} \) to be \((\hat{\alpha}_{iu} - \hat{\beta}_{iu})\) the above formula is the one that produces the item to user messages \( x_{i \rightarrow u}^{(k)} \) of the KOS algorithm. We reason likewise to prove the same for the update of worker’s messages. To prove the inductive step, we follow the same path as above to derive the same formula, and instead of initializing we use the induction hypothesis which states \( y_{u \rightarrow i}^{(k)} = (\hat{\alpha}_{iu} - \hat{\beta}_{iu}) \)

Thus, the messages of KOS algorithm and consequently its final output is fully recoverable by executing the iterative updates of the beta-shaped distribution algorithm.

We can obtain an analogous reformulation of the original KOS algorithm 3.1 by sending \((\alpha, \beta)\) pairs as messages, in place of single quantities \( x \) and \( y \), exchanging \( \alpha \) and \( \beta \) whenever \( a_{ui} < 0 \).

### 3.2.6 Limitations of KOS approach

The above re-statement of the KOS algorithm in terms of beta distribution updates sheds light on some of the limitations of the KOS algorithm.
Non-regular graphs. In real world scenarios, it is rare that items and workers are connected in a regular graph. Usually, some workers are very active providing multiple reviews, while others may provide only a few. Similarly, items have different popularity or visibility, some of them receiving many reviews, while others receiving only a few. In many real cases, power-law phenomena are in place.

The KOS algorithm may not perform well on non-regular graphs. To understand the reason, note that as the number of iteration progresses, the values of $x, y$ (or $\alpha$ and $\beta$ in our restatement) grow with a geometric rate related to the degrees of the nodes. Consider an item $i$, which is connected in the bipartite graph to two users, $u_1$ and $u_2$. The user $u_1$ is part of a large-degree subgraph; the user $u_2$ is instead part of a subgraph where all nodes (items and users) have small degree. Assume that $u_1$ and $u_2$ both give the same judgement (say, $+1$) about $i$. If the algorithm determines that $u_1$ has high reputation ($\alpha_{u_1} \gg \beta_{u_1}$), this reputation will be reflected in a strong certainty that the value of $i$ is $+1$ ($\alpha_i \gg \beta_i$). In the subsequent iteration from items to users, we will have that the full amount of $\alpha_i$ will be added to $\alpha_{u_2}$: the certainty of $u_1$ being truthful will be transferred to $u_2$. But this is of course inappropriate: $u_2$’s vote on $i$ is only one instance of agreement with the highly-reputed user $u_2$, and we should infer only a limited amount of certainty from one instance of behavior. In general, if the bipartite review graph is non-regular, the KOS algorithms will weigh excessively the evidence from the higher-degree portions of the graph. Our simulation results on artificial graphs will show that both the harmonic and the parameter estimation algorithms we propose outperform KOS on non-regular graphs.
Source dependence, and iterations over a finite graph. The additive nature of the KOS update rule makes the algorithm highly sensitive to the assumption of independent sources, and independence can fail, for two reasons.

First, the original sources (the users) are usually not statistically independent. For example, to answer a question such as “what is the phone number of this restaurant”, most workers will consult a limited number of sources such as Google Maps, Bing, and Yelp, and choose the source they trust the most in case of conflict. The workers would not be performing statistically independent inferences on the phone number; rather, they would be influenced by their a-priory trust in the information sources. The issue of crowds deriving their information from a limited number of information sources has been studies in finance; [31] show that the resulting correlation can hinder the ability of groups to make accurate aggregate predictions.

Furthermore, and even more relevant to our context, statistical dependence is generated simply by running the KOS algorithm on a finite graph for many iterations. Indeed, if the graph has degree \( m \), after \( n \) iterations we would need \((m - 1)^n\) distinct sources for them to independently contribute to the value at a node. This is analogous to the fact that most of our ancestors \( n > 30 \) generations ago appear in multiple nodes of our genealogical trees, since there were several orders of magnitude fewer than \( 2^n \) humans at that time. In essence, for each item, the infinite tree of inference with branching \( m - 1 \) is being folded inside the finite graph, and correlated information (corresponding to the same actual nodes in the graph) is being treated as if coming from independent graph nodes. The upshot is that after the first few initial rounds,
the updates recycle the same information. Indeed, our experiments show that on finite graphs, the performance of KOS and other iterative methods peaks after a few initial rounds, and gets worse as the method reaches the fixed point. These empirical results appear to contradict the optimality results given in [34], but the contradiction is only apparent. The optimality results proved in [34] concern the behavior when the number of reviewers, and the size of the graph, grow to infinity; they do not concern the problem of optimally extracting information from a finite graph.

Our proposed algorithms are also affected by source correlation. However, our empirical results indicate that they are less affected than KOS. Intuitively, this is because our updates are performed based on reputation mean, rather than adding up the shape parameters.

3.3 Two proposed algorithms: Regularized Harmonic and Beta Shape Parameter Estimation

We now describe two methods for the binary crowdsourced labels aggregation problem. Both algorithms model the distributions of item quality and user reliability via beta distributions, updating the distributions in iterative manner.

The *Regularized Harmonic Algorithm* is derived from the beta-distribution interpretation of KOS by adopting an update rule based on distribution means, rather than addition of shape parameters. This leads to a simple and efficient algorithm that performs well in presence of correlated information.
The \textit{Beta Shape Parameter Estimation Algorithm} uses beta distributions to represent both item and worker distributions, and performs updates by first performing a Bayesian update, and then using parameter estimation to approximate the posterior distributions with beta distributions.

In both algorithms, we assume that each item is associated with a quality or ambiguity $y$ that corresponds to a Bernoulli trial probability of the item being perceived as true by a perfectly reliable user. Similarly, each user has a probability $x$ of telling the truth (i.e. report accurately the result of the bernulli trial of the item), and $1 - x$ of lying (i.e., reporting the opposite of the observed result). We assume that $y$ and $x$ follow distributions that can be approximated by beta distributions.

The root reason why these algorithms outperform EM is that unlike EM, the algorithms explicitly represent (via the variance of the beta distributions) the amount of information we have on each user and item, so that they can distinguish users with the same average quality but different amounts of certainty over it.

\subsection{Regularized Harmonic Algorithm}

The \textit{Regularized Harmonic Algorithm} represents the knowledge about a user $u$ via a beta distribution $\text{Beta}(\alpha_u, \beta_u)$, and the knowledge about an item $i$ via a beta distribution $\text{Beta}(\alpha_i, \beta_i)$. The update rule (3.2) adds the shape parameters $\alpha_u, \beta_u$ of users $u \in \partial i$ to compute the shape parameters of item $i$. Thus, a user $u$ whose distribution has shape parameters $\alpha_u, \beta_u$ has an influence proportional to $\alpha_u + \beta_u$. As $\alpha_u$ and $\beta_u$ grow during the iterations, this can affect the performance over non-regular
graphs, and in presence of correlated information, as discussed earlier. In the harmonic algorithm, the influence of a user is proportional to $|2p_u - 1|$, where $p_u = \alpha_u / (\alpha_u + \beta_u)$ is the mean of the beta distribution; and symmetrically for items. This leads to a more stable update rule, where differences in graph degree, and information correlation, have a more moderate effect on the final result. The detailed algorithm is given in Figure 3.2, where we use the standard notation $x^+ = (x + |x|)/2$ for the positive part of $x$.

### 3.3.2 Beta Shape Parameter Estimation Algorithm

The Beta Shape Parameter Estimation Algorithm, which we abbreviate by BSP, also models user and item distributions as beta distributions. The algorithm updates iteratively these distributions by first performing a pure Bayesian update, obtaining general posterior distributions, and then by re-approximating these posterior distributions by beta distributions having the same mean and variance. The idea of making a specific assumption about a distribution and performing approximate bayesian updates using parameter estimation is fairly classical; it was applied in a crowdwourcing context in [23] to the problem of computing chess and tennis player rankings from match outcomes making a normal distribution assumption for the strengths of the players.

In practice, BSP never computes the actual posterior distributions; these distributions are simply used as a mathematical device to derive update rules that are expressed directly in terms of shape parameters. To derive the update, assume that user $u$ voted True (or Yes) for item $i$. We assume there is a prior for the quality of the item, given by distribution Beta($\alpha_i, \beta_i$). We can observe the event of a True vote by
**Input**: Bipartite graph $E \subseteq U \times I$, answers $a_{ui}, k_{\text{max}}$

**Output**: Estimation of correct solutions $s_i$ for all $i \in I$.

```plaintext
1 foreach user $u$ and item $i$ do
  $\alpha_u = 1 + \Delta$ for some $\Delta > 0$, and $\beta_u = \alpha_i = \beta_i = 1$.
2 for $k = 1, \ldots, k_{\text{max}}$ do
  foreach user $u \in U$ do $p_u \leftarrow \alpha_u / (\alpha_u + \beta_u)$
  foreach item $i \in I$ do
    $\alpha_i \leftarrow 1 + \sum_{u \in \partial i} (a_{ui}(2p_u - 1))^+$
    $\beta_i \leftarrow 1 + \sum_{u \in \partial i} (-a_{ui}(2p_u - 1))^+$
  foreach item $i \in I$ do $p_i \leftarrow \alpha_i / (\alpha_i + \beta_i)$
  foreach user $u \in U$ do
    $\alpha_u \leftarrow 1 + \sum_{i \in \partial u} (a_{ui}(2p_i - 1))^+$
    $\beta_u \leftarrow 1 + \sum_{i \in \partial u} (-a_{ui}(2p_i - 1))^+$
4 Return estimate vector $\hat{s}_i = \text{sign}(\alpha_i - \beta_i)$ for all $i \in I$.
```

Figure 3.2: Regularized Harmonic Algorithm.
on $i$ when one of two mutually exclusive events occur: either the item was seen as true and the user reported it as true, or the item was seen as false, but the user flipped the observation. The probability of the former event is $x \cdot y$, and the probability of the latter is $(1 - x) \cdot (1 - y)$; given that the two events are mutually exclusive, the overall probability of a vote True is their sum. A Bayesian update for the item distribution after the user True vote yields:

$$g^{(k+1)}(y) \propto g^{(k)}(y) \int_0^1 (x \cdot y + (1 - x) \cdot (1 - y)) \cdot x^{\alpha_u - 1} \cdot (1 - x)^{\beta_u - 1} dx$$  \hspace{1cm} (3.3)

BSP starts by assigning a prior to the users reputations. The choice of the prior is open. In most cases, we use a prior where users are considered weakly truthful, corresponding to a beta distribution with shape parameters $\alpha = 1 + \Delta$ and $\beta = 1$, where $\Delta > 0$ is small.

We use the votes of the users to update the items distribution by calculating the bayesian update through integration and normalization of (3.3). The derived function is not a beta distribution and further updates in an iterative manner are intractable. We thus approximate the derived distribution by beta distribution through calculation of the expectation and variance of the derived distribution and estimation of the shape parameters of the beta distribution that corresponds to this expectation and variance.

Below, we provide the details on performing the parameter estimation used in BSP.
3.3.3 Obtaining the bayesian updates

The indefinite integral of the Bayesian derivation of (3.3) can be written as follows, where $B_x$ stands for the incomplete Beta function:

\[
F(x) = \int (x \cdot y + (1 - x) \cdot (1 - y)) \cdot x^{\alpha_u - 1} \cdot (1 - x)^{\beta_u - 1} \, dx
\]

\[
= \int (2xy - x - y + 1) \frac{dB_x(\alpha_u, \beta_u)}{dx} \, dx
\]

\[
= (2xy - x - y + 1)B_x(\alpha_u, \beta_u) - (2y - 1) \int B_x(\alpha_u, \beta_u) \, dx
\]

\[
= (2xy - x - y + 1)B_x(\alpha_u, \beta_u) - (2y - 1)(xB_x(\alpha_u, \beta_u) - B_x(\alpha_u + 1, \beta_u))
\]

Thus, the definite integral in the $[0,1]$ interval is:

\[
F(1) - F(0) = F(1) = yB(\alpha_u, \beta_u) - (2y - 1)(B(\alpha_u, \beta_u) - B(\alpha_u + 1, \beta_u))
\]

\[
= B(\alpha_u, \beta_u) \left[ y \left( \frac{\alpha_u}{\alpha_u + \beta_u} - 1 \right) + \left( 1 - \frac{\alpha_u}{\alpha_u + \beta_u} \right) \right]
\]

\[
= y \left( \frac{\alpha_u}{\alpha_u + \beta_u} - 1 \right) + \left( 1 - \frac{\alpha_u}{\alpha_u + \beta_u} \right).
\]

Letting $p_u = \alpha_u / (\alpha_u + \beta_u)$ we can rewrite the Bayesian update as:

\[
g^{(k+1)}(y) \propto g^{(k)}(y) \cdot (y(2p_u - 1) + (1 - p_u)).
\]

With a similar reasoning we obtain the derivation when the user votes that item $i$ is false (that is, $a_{ui} = -1$). The probability of that event is now $x(1 - y) + (1 - x)y$ and the integrated function is:

\[
h(x) = (x(1 - y) + (1 - x)y)x^{\alpha_u - 1} \cdot (1 - x)^{\beta_u - 1}.
\]
A similar integration leads to the derivation:

$$g(k+1)(y) \propto g(k)(y) \cdot (y(1 - 2p_u) + p_u) .$$

Thus, the general expression that contains user’s answer $a_{ui}$ as a parameter is:

$$g(k+1)(y) \propto g(k)(y) \cdot \left[ y(2p_u - 1)a_{ui} + \left( \frac{1}{2} - p_u \right) a_{ui} + \frac{1}{2} \right]$$

(3.4)

We assume that the item quality before the update was also a beta distribution, with shape parameters $\alpha_i$ and $\beta_i$. In this case, the normalization constant of the updated distribution $g(y)$ is a function of the shape parameters $\alpha_i, \beta_i$ and $p_u$ as follows:

$$\Xi(\alpha_i, \beta_i, p_u) = \int_0^1 g(y)dy = \int_0^1 (2p_u - 1)a_{ui}y^{\alpha_i}(1 - y)^{\beta_i - 1} + \left[ \left( \frac{1}{2} - p \right) a_{ui} + \frac{1}{2} \right] y^{\alpha_i - 1}(1 - y)^{\beta_i - 1}dy$$

$$= (2p_u - 1)a_{ui}B(\alpha_i + 1, \beta_i) + \left[ \left( \frac{1}{2} - p_u \right) a_{ui} + \frac{1}{2} \right] B(\alpha_i, \beta_i).$$

The precise derivation of $g(k+1)$ (as a function of $y, p_u, \alpha_i, \beta_i$) is:

$$\frac{(2p_u - 1)a_{ui}y^{\alpha_i}(1 - y)^{\beta_i - 1} + \left( \left( \frac{1}{2} - p_u \right) a_{ui} + \frac{1}{2} \right) y^{\alpha_i - 1}(1 - y)^{\beta_i - 1}}{(2p_u - 1)a_{ui}B(\alpha_i + 1, \beta_i) + \left( \left( \frac{1}{2} - p_u \right) a_{ui} + \frac{1}{2} \right) B(\alpha_i, \beta_i)}. $$

3.3.4 Parameter estimation for the beta distribution

Making the binomial assumption on the quality of items, we can approximate the derived distribution of the item quality with a beta distribution by estimating its shape parameters. The expectation of random variable $y$ after the derivation is:

$$E[y] = \int_0^1 yg(y)dy$$

$$= \frac{(2p_u - 1)a_{ui}}{\Xi(\alpha_i, \beta_i, p_u)} \int_0^1 y^{\alpha_i + 1}(1 - y)^{\beta_i - 1}dy + \frac{1 - p_u}{\Xi(\alpha_i, \beta_i, p_u)} \int_0^1 y^{\alpha_i}(1 - y)^{\beta_i - 1}dy$$

$$= \frac{(2p_u - 1)a_{ui}}{\Xi(\alpha_i, \beta_i, p_u)} B(\alpha_i + 2, \beta_i) + \frac{\left( \left( \frac{1}{2} - p_u \right) a_{ui} + \frac{1}{2} \right)}{\Xi(\alpha_i, \beta_i, p_u)} B(\alpha_i + 1, \beta_i)$$
We denote the above expression as \( e(\alpha_i, \beta_i, p_u) \) To calculate the variance we first compute the expectation of \( y^2 \):

\[
E[y^2] = \int_0^1 y^2 g(y) dy = \frac{1}{\Xi(\alpha_i, \beta_i, p)} B(\alpha_i + 3, \beta_i) + \left( \frac{p - 1}{p} \right) a_{ui} + \frac{1}{2} B(\alpha_i + 2, \beta_i)
\]

We denote the expression of \( E[y^2] \) as function \( s(\alpha_i, \beta_i, p_u) \). The variance, as a function of \( \alpha_i, \beta_i \) and \( p_u \) is thus:

\[
v(\alpha_i, \beta_i, p_u) = s(\alpha_i, \beta_i, p_u) - e^2(\alpha_i, \beta_i, p_u)
\]

We can estimate the beta distribution of the item, as the beta distribution that has the same expectation and variance. Denoting the shape parameters of that beta distribution as \( \alpha'_i \) and \( \beta'_i \) and expression \( \frac{1 - e(\alpha_i, \beta_i, p_u)}{e(\alpha_i, \beta_i, p_u)} \) as \( t(\alpha_i, \beta_i, p_u) \) we have:

\[
\frac{\alpha'_i}{\alpha'_i + \beta'_i} = e(\alpha_i, \beta_i, p_u)
\]

\[
\beta'_i = \alpha'_i \frac{1 - e(\alpha_i, \beta_i, p_u)}{e(\alpha_i, \beta_i, p_u)} \cdot \alpha'_i = t(\alpha_i, \beta_i, p_u) \cdot \alpha'_i
\]  

(3.5)

Using the variance \( v(\alpha_i, \beta_i, p_u) \) of the derived distribution as the variance of the beta distribution we have:

\[
v(\alpha_i, \beta_i, p_u) = \frac{\alpha'_i \beta'_i}{(\alpha'_i + \beta'_i)^2 (\alpha'_i + \beta'_i + 1)}
\]

\[
 = \frac{\alpha^2 t(\alpha_i, \beta_i, p_u)}{(t(\alpha_i, \beta_i, p_u) + 1)^2 \alpha^2 \left( (t(\alpha_i, \beta_i, p_u) + 1) \alpha'_i + 1 \right)}
\]

\[
 = \frac{t}{(t(\alpha_i, \beta_i, p_u) + 1)^2 \left( \alpha'_i (1 + t(\alpha_i, \beta_i, p_u)) + 1 \right)}
\]

\[
v(\alpha_i, \beta_i, p_u) = \frac{e^2 \cdot t}{1 + \alpha'_i / e} = \frac{e^3 \cdot t}{\alpha'_i + e} = \frac{e^2 \cdot (1 - e)}{\alpha'_i + e}
\]

\[
\alpha'_i = \frac{e^2 \cdot (1 - e)}{v} - e
\]  

(3.6)
Having obtained $\alpha_i'$ we can use (3.5) to obtain the $\beta_i'$ shape parameter of the approximated beta distribution.

3.3.5 Extention to the case of multiple votes

In most real scenarios, each item receives multiple votes from different users and each user casts many votes across different items. One way to handle this in the framework of our parameter estimation method is to pick one user, obtain the derivation of the posterior distribution of the item quality given the user vote, approximate a new beta distribution for the user quality distribution and continue the same procedure for the remaining user votes sequentially. This approach suffers for two reasons: it is not deterministic, as our choice of the users votes order affects the outcome and, multiple approximations can have a negative impact on the quality. A preferred approach is to obtain the precise derivation given all the users votes, and then approximate the derived user quality distribution once. We assume that $m$ users have voted on each item, and that each user $u$ is associated with a different reputation distribution $r_u(x)$ which is a beta distribution with shape parameters $\alpha_u$ and $\beta_u$. Out of the $m$ users we assume that $t \leq m$ users have voted that item $i$ is true, and the rest $m - t$ users voted that the item is false. An additional assumption is independence: each user casts a vote independently on what other users have voted. The Bayesian update of the item distribution is thus:

$$g^{(k+1)}(y) \propto g^{(k)}(y) \int_{x_1=0}^1 \cdots \int_{x_m=0}^1 \prod_{u=1}^t (x_u y + (1 - x_u)(1 - y)) r_u(x_u) \cdot \prod_{u=t+1}^m ((1 - x_u)y + x_u(1 - y)) r_u(x_u) \prod_{u=1}^m dx_u.$$
Each factor of the integrated product contains a different variable $x_u$ as well as $y$ which does not appear in the differentials and is thus treated as a constant for purposes of integration. The factors can be completely separated in a product of integrals:

$$g^{(k+1)}(y) \propto g^{(k)}(y) \prod_{u=1}^{t} \int_{x_u=0}^{1} (x_u y + (1 - x_u)(1 - y)) r_u(x_u) dx_u \cdot \prod_{u=t+1}^{m} \int_{x_u=0}^{1} ((1 - x_u)y + x_u(1 - y)) r_u(x_u) dx_u$$

Using (3.4) the above product is:

$$g^{(k+1)}(y) \propto g^{(k)}(y) \cdot \prod_{u \in \partial i} (y(2p_u - 1)a_{ui} + \frac{1}{2} - p_u)a_{ui} + \frac{1}{2}$$

$$= g^{(k)}(y) \prod_{u \in \partial i} (2p_u - 1)a_{ui} \cdot \prod_{u \in \partial i} y + \frac{1}{2} - p_u)a_{ui} \cdot \frac{1}{(2p_u - 1)a_{ui}}$$

$$\propto g^{(k)}(y) \prod_{u \in \partial i} y + \frac{1}{2} - p_u)a_{ui} + \frac{1}{2} \cdot \frac{1}{(2p_u - 1)a_{ui}} \tag{3.7}$$

where $p_u$ is the mean of the user’s distribution reputation, i.e. $p_u = \alpha_u / (\alpha_u + \beta_u)$. The product is a polynomial of $y$ of degree $m = |\partial i|$ with roots

$$\frac{1}{2} - p_u)a_{ui} + \frac{1}{2} \cdot \frac{1}{(2p_u - 1)a_{ui}}$$

for each user $u$. Finding the normalization constant for the above distribution requires us to calculate the integral of the polynomial. A natural approach is to obtain the monomial form of the polynomial and trivially obtain the integral as the sum of the integrals of its monomials. Expanding the factors into monomials leads to a number of monomials that is exponential to the number of the roots that we can sum to obtain the monomial form of the polynomial. However, there is a computationally efficient way to obtain the monomial form of a polynomial from its roots. Assume that a polynomial
\( p(y) \) is in form \((x - r_1) \cdots (x - r_m)\). We define \( p_j(y) \) to be the polynomial containing the first \( j \) factors, for instance, \( p_2(x) = (x - r_1) \cdot (x - r_2) \). We can obtain the coefficients of the monomial form of the polynomial \( p(y) \) incrementally in the following way:

\[
p_{j+1}(y) = p_j(y) \cdot (y - r_{j+1}) = y \cdot p_j(y) - r_{j+1} \cdot p_j(y)
\]

At each iteration, every coefficient \( c_d \) of monomial of degree \( d \) of the polynomial is updated as shown below:

\[
c_d' = c_{d-1} - r_j \cdot c_d \cdot r_j.
\]

Doing the same operation for each one of the \( m \) coefficients, and for at most \( m \) iterations, we can obtain the monomial form of the polynomial in \( O(m^2) \) steps. Applying that procedure to (3.7), we obtain a polynomial and the derivation is:

\[
g^{(k+1)}(y) \propto g^{(k)}(y) \cdot (c_m y^m + \cdots + c_1 y + c_0).
\]

Note that in all cases \( c_m = 1 \). Again, assuming that the item quality distribution \( g(y) \) is a beta distribution with parameters \( \alpha_i \) and \( \beta_i \) the derivation becomes:

\[
g^{(k+1)}(y) \propto \sum_{u=0}^{m} c_u y^{u+\alpha_i-1} \cdot (1 - y)^{\beta_i-1}.
\]

We calculate the normalization constant of the distribution as follows:

\[
\Xi(\alpha_i, \beta_i, \vec{c}) = \int_0^1 g^{(k+1)}(y) dy = \int_0^1 \sum_{u=0}^{m} c_u y^{u+\alpha_i-1} \cdot (1 - y)^{\beta_i-1} dy
\]

\[
= \sum_{u=0}^{m} \int_0^1 c_u y^{u+\alpha_i-1} \cdot (1 - y)^{\beta_i-1} dy
\]

\[
= \sum_{u=0}^{m} c_u \cdot B(u + \alpha_i, \beta_i).
\]
Similarly to the one user case, we calculate the expectation of random variable $y$ after the derivation:

$$E[y] = \frac{1}{\Xi(\alpha_i, \beta_i, \vec{c})} \int_{0}^{1} yg(y)dy = \frac{1}{\Xi(\alpha_i, \beta_i, \vec{c})} \int_{0}^{1} \sum_{u=0}^{m} c_u y^{u+\alpha_i} \cdot (1 - y)^{\beta_i - 1}dy$$

$$= \frac{1}{\Xi(\alpha_i, \beta_i, \vec{c})} \sum_{u=0}^{m} \int_{0}^{1} c_u y^{u+\alpha_i} \cdot (1 - y)^{\beta_i - 1}dy$$

$$= \frac{1}{\Xi(\alpha_i, \beta_i, \vec{c})} \sum_{u=0}^{m} c_u \cdot B(u + \alpha_i, 1, \beta_i).$$

The expectation of $y^2$ is thus:

$$E[y^2] = \frac{1}{\Xi(\alpha_i, \beta_i, \vec{c})} \sum_{u=0}^{m} c_u \cdot B(u + \alpha_i + 2, \beta_i).$$

Given the two above quantities we can proceed to estimate parameters $\alpha'_i$ and $\beta'_i$ of the beta distribution that approximates $g(y)$ using (3.6) and (3.5) in an identical way to the one user case. After completing the updates, the BSP method will use the new item distributions along with user answers to update the user distributions in an identical symmetric manner.

The procedure proceeds in a symmetrical way to update the user distributions from the item distribution and the votes. BSP performs these iterative updates of user and item distributions, either until the distributions converge (the difference across iterations is below a specified threshold), or until we reach a desired number of iterations. After the final iteration, we label with True (or Yes) the items $i$ for which $\alpha_i \geq \beta_i$, and we label with False (or No) the others.

**Excluding self-votes.** Similarly to the KOS method [34], for each user we can have separate distributions $g_{ui}$ for all the items $i$ the user voted on, where the distri-
bution $g_{ui}$ represents the information on $u$ derived without using $i$ directly; similarly for the distributions $r_{ui}$ for items. The final estimate of item labels is obtained by adding all the differences $\alpha - \beta$ of the shape parameters of all the distributions for the item, and obtaining the sign. We provide a pseudocode for the method in Figure 3.3.

### 3.4 Experimental study on synthetic data

We conducted experiments on synthetic regular graphs as in [34] and [38]. We also tested graphs whose vertex degrees follow a uniform distribution in a predefined range, and graphs whose vertex degrees follow a Pareto distribution. This is true in many graphs where task allocation is not defined by the algorithm: for example, popular items on Yelp will have much more votes than others, following some form of power law.

#### 3.4.1 Independent users model

In the spammer-hammer model of [34], users are either spammers or honest. Spammers provide random answers, that is, they reply True or False with 50% probability regardless of the true label, while honest workers report the truth. We also use a model where user accuracies follow a uniform distribution in $[0.5, 1]$, and a model where user accuracies follow a beta distribution with $\alpha = 0.03, \beta = 0.01$ which corresponds to a mean accuracy of $0.75$ and variance $\approx 0.18$. Parameter $q$ represents the percentage of honest workers. We report the fraction of misclassified labels, averaged for 10 runs of the algorithm (on newly constructed graphs) which is an estimate of the probability of misclassification for a given method. For sets with balanced classes, the average er-
Input: bipartite graph $E \subseteq U \times I$, answer set $A, k_{\text{max}}$

Output: Estimation of correct solutions $s : \hat{s}(\{a_{ui}\})$

1. foreach $(u, i) \in E$ do
   
   2. Initialize user distributions $r_{ui}(x)$ with Beta$(1 + \Delta, 1)$ for some $\Delta > 0$;
   
   3. Initialize item distributions $g_{ui}(y)$ with Beta$(1, 1)$;

4. for $k = 1, \ldots, k_{\text{max}}$ do
   
   5. foreach item $i \in I$ do
      
      6. foreach item $u \in \partial i$ do
         
         7. Obtain $g_{ui}^{(k+1)}(y)$ through pure bayesian derivation;
         
         8. Obtain $\alpha_{ui}^i$ and $\beta_{ui}^i$ for $g_{ui}$ through shape parameter estimation; $g_{ui}^{(k+1)}(y) \leftarrow \text{Beta}(\alpha_{ui}^i, \beta_{ui}^i)$;

      9. foreach user $u \in U$ do
         
         10. foreach item $i \in \partial u$ do
             
             11. Obtain $r_{ui}^{(k+1)}(y)$ through pure bayesian derivation;
             
             12. Obtain $\alpha_{ui}^u$ and $\beta_{ui}^u$ for $r_{ui}$ through shape parameter estimation; $r_{ui}^{(k+1)}(y) \leftarrow \text{Beta}(\alpha_{ui}^u, \beta_{ui}^u)$;

   13. Return vector $\hat{s}_i = \text{sign}(\sum_{u \in \partial i} \alpha_{ui}^i - \beta_{ui}^i)$ for all $i \in I$.

Figure 3.3: Beta Shape Parameter estimate (BSP) algorithm.
ror is a reliable performance measure. We also conduct experiments with varying class balance skew and report the F-1 measure which is more appropriate in this case.

3.4.2 The limited sources model

In one set of experiments, we induce correlation to users’ answers by forcing them to seek their answers by a limited set of sources. We fix the number of sources to 5, the sources vote on the items with a predefined accuracy, then users pick one of the sources to seek the answer in the following manner: the most popular source is picked with a given probability, the second most probable source is picked with 20% less probability and likewise for the remaining sources. This depicts a realistic scenario where some internet sources are more popular than others or contain more information. The correlation induced in this manner can be a challenge for algorithms that aim to obtain reliable aggregations from crowdsourcing. We could get overconfident in an answer which is in fact only a replicated erroneous answer from a single source.

3.4.3 Algorithms

We implement the BSP approach, the Harmonic approach, EM [12], KOS approach [34], and AMF-EM [38]. The two last methods are the state-of-the-art methods of the literature. We also implemented a variation of the KOS method suitable for non-regular graphs (uniform and Pareto) which we call ‘Regularized KOS algorithm’. It works in the same manner but regularizes item messages by the square root of the total votes they received (for and against). In that way, it prevents items becoming
Figure 3.4: Results for a 100×100 regular bipartite graph with 5 votes per item using the spammer-hammer model. The percentage of ‘hammers’ in the crowd, that is, accurate users that answer truthfully, is 60%, while the rest 40% are spammers that give random answers, that is, they return ‘true’ or ‘false’ each with probability 50% regardless of the true label of items.

very highly reputed simply because they have received more votes than average. In all synthetic cases, we have a prior on user’s reputation that is slightly truthful with $\Delta = 0.001$, that is, we are almost agnostic with respect to human workers reliability.

**Statistical significance testing.** We do not know the distribution of the average error or other performance measures of the algorithms. However, the result of independent runs across different random graphs are i.i.d. random variables. For large samples (≥ 30), due to the central limit theorem, the arithmetic mean of a sufficiently large number of iterates of independent random variables is approximately normally distributed, regardless of the underlying distribution, and the $z$-test is an appropriate statistical hypothesis test method in this case [54]. We conducted the $z$-test for large samples (≥ 80) across different runs, using the unbiased sample variance to approximate
the distribution variance, to confirm the relevance of results that we see in the plots.

When we report a superior result, we have confirmed its statistical significance with a high enough sample size that makes the test reject the null hypothesis with very low critical values (<< 0.01), i.e. very high confidence.

3.4.4 Results

Figure 3.4 shows the results for a regular bipartite graph of 100 users and 100 items for levels of spamming at 40% ($q = 0.6$) using the spammer-hammer model. We omit results on majority voting when it is outperformed by all iterative methods (the plot already contains several methods). Figure 3.5 shows results for uniformly distributed user accuracies in the range [0.5,1], and Figure 3.6 shows results using the beta distribution to model user accuracies with $\alpha = 0.03$ and $\beta = 0.01$. These shape parameters correspond to a mean of 0.75 and a variance $\approx 0.18$. We observe the supe-
riority of the BSP and Harmonic approaches, confirmed through significance testing, to other techniques while we note that simple majority voting outperforms EM in Figure 3.5, and both EM and EM-AMF in Figure 3.6.

Figure 3.7 shows the comparative performance using the limited sources model. The number of sources is 5. The sources vote on the items with a predefined accuracy of 60%, then users pick one of the sources to seek the answer according to a distribution that makes every source 20% more likely to be picked than the next more popular source. We construct this distribution by assigning a value to the most popular source, then set the value of the second most popular source to 80% of the value of the most popular source, and continue with the same relative probability for all sources. In the end, we normalize accordingly to make the values of all sources sum up to 1.

Again, the BSP method and the harmonic approach demonstrate the best performance, yet, the harmonic approach is the one performing better. The BSP method observably deteriorates as the iterations increase, reinforcing errors due to the independence assumption which does not hold.

Figure 3.8 shows the performance for a graph where the number of votes on each item follow a Pareto distribution with shape parameter 0.9. We used both KOS standard and regularized approaches. EM-AMF, BSP and harmonic outperform the rest with a slight lead of BSP. KOS approach performs poorly, the regularized variant we implemented provided benefits but the performance is still low. We also confirm that the KOS method diverges from its best performance as iterations increase when the regularity assumption is broken.
Finally, Figures 3.9 and 3.10 show the performance for a graph that is $1000 \times 1000$, an order of magnitude higher in the number of nodes for the beta distribution accuracies and limited sources models. We confirm the superiority of both the BSP and Harmonic methods in Figure 3.9 and that of the BSP method in Figure 3.10.

We omit results for graphs with uniform distribution of edges which demonstrate a similar performance.

Figure 3.11 shows the performance with increasing votes per item. Unsurprisingly, majority voting steadily improves with increasing number of votes (as the law of big numbers materializes and the percentage of inaccurate responses converges to their probability). KOS significantly improves with increasing number of votes and matches BSP and Harmonic in absolute correctness for 13 and 15 votes per item. Surprisingly, 9 votes per item are worse than 7 votes per item for all three iterative methods (and we confirmed its statistical significance). The reasons for this are not evident. An explanation is that while 9 and 11 votes are more than 7, a graph with 9 and 11 votes per item has a wider span and an error may propagate more widely compared to a graph with fewer votes. For the case of 7, the graph hits a sweet spot where the votes are high enough that it improves over fewer votes, but also few enough so the impact of errors remains localized. A formal explanation of the phenomenon may be an object of further study.

Figure 3.12 shows the F1 measure as we vary the skew of the balance of classes. BSP and Harmonic (alternately) have the lead in performance as skew varies, with Harmonic demonstrating less sensitivity across the variance of the skew.
Figure 3.6: Results for a 100×100 regular bipartite graph with beta distributed user accuracies having $\alpha = 0.03, \beta = 0.01$. These shape parameters correspond to a mean user accuracy of 0.75 and variance $\approx 0.18$. A beta distribution is a more realistic representation of human worker accuracies in a crowd than the uniform assumption or the spammer-hammer model.

3.5 Experimental study on real-world data

To demonstrate the relevance of our approach on real-world data, we conducted experiments with datasets obtained through Amazon’s Mechanical Turk. For the workers priors, we make the assumption they are 4 times more likely to be reliable than not, that is, $\Delta = 3$.

Statistical significance testing. We sample the edges of the bipartite graph that corresponds to the real world dataset with a probability of 90%, constructing a set of random subgraphs. For each of the random subgraphs we obtain the required performance using two different methods measure and run the $z$-test on the results to determine statistical significance of difference in performance between the methods.
Figure 3.7: Results for a 100×100 regular bipartite graph with the limited sources model (5 sources). The number of sources is 5. The sources vote on the items with a predefined accuracy of 60%, then users pick one of the sources to seek the answer according to a distribution that makes every source 20% more likely to be picked than the next more popular source. We construct this distribution by assigning a value to the most popular source, then set the value of the second most popular source to 80% of the value of the most popular source, and continue with the same relative probability for all sources. We normalize accordingly to make the values of all sources sum up to 1. This model represents a realistic scenario where human workers consult a limited number of sources to obtain answers, and each source has different popularity and hence, a different probability of being picked.
Figure 3.8: Results for a 100 x 100 Pareto distribution graph with the limited sources model (5 sources). The degrees of the nodes of the graph follow a Pareto distribution (power law) with shape parameter 0.9. The power law ensures that some items have low number of votes (minimum 5) and few items have potentially very high number of votes. This represents a scenario where the tasks are not allocated to human workers by the algorithm but workers freely assign votes to items. The number of sources is 5. The sources vote on the items with a predefined accuracy of 60%, then users pick one of the sources to seek the answer according to a distribution that makes every source 20% more likely to be picked than the next more popular source.
Figure 3.9: Results for a 1000×1000 regular bipartite graph with beta distributed user accuracies having $\alpha = 0.03$, $\beta = 0.01$. These shape parameters correspond to a mean user accuracy of 0.75 and variance $\approx 0.18$. A beta distribution is a more realistic representation of human worker accuracies in a crowd than the uniform assumption or the spammer-hammer model. The plot shows the comparative performance for a large structure of 1000 users $\times$ 1000 items (as opposed to Figure 3.6 which reports results for the same model on a smaller 100 $\times$ 100 graph.)
Figure 3.10: Results for a $1000 \times 1000$ Pareto distribution graph with the limited sources model (5 sources). The degrees of the nodes of the graph follow a Pareto distribution (power law) with shape parameter 0.9. The power law ensures that some items have low number of votes (minimum 5) and few items have potentially very high number of votes. This represents a scenario where the tasks are not allocated to human workers by the algorithm but workers freely assign votes to items. The number of sources is 5. The sources vote on the items with a predefined accuracy of 60%, then users pick one of the sources to seek the answer according to a distribution that makes every source 20% more likely to be picked than the next more popular source. The plot shows the comparative performance for a large structure of 1000 users $\times$1000 items (as opposed to Figure 3.8 which reports results for the same model on a smaller 100 $\times$ 100 graph.)
Figure 3.11: Comparative performance with increasing number of votes per item. Each user has a fixed accuracy and user accuracies follow a beta distribution having $\alpha = 0.03$, $\beta = 0.01$. These shape parameters correspond to a mean user accuracy of 0.75 and variance $\approx 0.18$. The plots show that KOS, BSP and the Harmonic approach are able to reach zero error for $>13$ votes/item.

Figure 3.12: Comparative performance with varying skew of the two classes. Comparative performance with increasing number of votes per item. Each user has a fixed accuracy and user accuracies follow a beta distribution having $\alpha = 0.03$, $\beta = 0.01$. These shape parameters correspond to a mean user accuracy of 0.75 and variance $\approx 0.18$. The reported evaluation measure is the F1 measure of the positive class.
3.5.1 Wikipedia edits Mechanical Turk data

We tested our method on a real world dataset involving judgements of Mechanical Turk workers on a set of Wikipedia edits. The question that the workers addressed was whether a particular edit was valid or the result of vandalism. The set of workers and edits are therefore an instance of the binary crowdsourcing problem that our methods address. For this dataset, we do not have an explicit ground truth for the Wikipedia edits. We obtain the ground truth for a set of edits through the redundant nature of the data. For a particular set of edits we have a very high number of votes. We isolate the edits that have more than 25 votes and obtain what we deem as ground truth through majority voting. The number of votes is high enough that we can with high confidence consider that the ground truth for those edits can be obtained only through the consensus.
The idea is that majority voting approaches correctness as the number of votes increase. This is the same intuition used in [46] to measure the precision of crowdsourced grading algorithms. While we acknowledge that a small margin of error might remain after averaging 25 judgements, we argue that the residual error is likely to be small. Sadly, obtaining further information on what has been reverted on the Wikipedia is not practical and also not reliable. Edits can be reverted multiple times, and reversions do not always indicate bad content leading to disaffected editors [1]. Also, training ML models with independent set of parameters to detect vandalism converges to the golden set obtained through annotators [42], an additional indication that obtaining labels through highly redundant annotation from human workers is a reliable method.

To construct the bipartite graph, we obtain the earlier 5 votes for all the highly-voted items. These votes, and the corresponding workers and edits, together with the ground truth that we obtain by using all the votes, define the bipartite graph on which we run and test our algorithms. We end up with a graph of 1927 edits and 677 workers.

Figure 3.13 shows the results for the comparative iterations study for BSP, regularized Karger, Harmonic Bayesian, EM-AMF and plain majority voting.

A summary of the iterations study, all approaches start with similar or close results to majority voting, and then the error rates go higher as the iterations increase, apparently due to reinforcing wrong beliefs on some users reputation. BSP appears to be robust to this effect, and is the only method that has a lead over simple majority voting reducing the error rate of the vandalism classification. The difference with majority voting is small (in the order of 1%) but statistically significant.
Figure 3.14: Comparative performance for the RTE dataset.

Figure 3.15: Comparative performance for the temp dataset.
3.5.2 NLP Mechanical Turk datasets

We also conducted experiments with publicly available NLP datasets obtained through Amazon’s Mechanical Turk [53]. Unlike the Wikipedia dataset, for these datasets we have a given ground truth. Similarly to [38], we report results for increasing number of votes per task. We show the performance of the methods after 5 iterations. The results of figure 3.14 for the textual entailment dataset (RTE) confirm the divergence of the [34] approach and show that with the exception of KOS which performs bad, the iterative methods have generally a comparable performance. We omit the vanilla EM from the already dense plot as its performance is comparable.

We omit the KOS results from Figure 3.15 which shows the comparative performance on the Temp dataset. KOS diverges to almost 50% of error, so the figure omits it to zoom to the performance of the other iterative methods.
3.5.3 The Duchenne experiment dataset

The last real-world dataset we examined is the publicly available dataset from the Duchenne experiment which was used in [65], and contains binary answers from MTurk users on whether a face is smiling along with the ground truth. Performance is unstable in the first few iterations for all methods and stabilizes after several rounds. For the average error, regularized KOS is again diverging, while EM-AMF emerges as slightly superior to majority voting for the average error, also confirmed through significance testing.

3.5.4 Average recall for real-world datasets

The results on the real-world datasets are inconclusive with respect to which method is superior, in line with other studies [51] that show performance of methods oscillating across datasets. Average recall has emerged as an efficient evaluation measure where the data has a skewed class distribution. F-1 measure is often an appropriate measure but it is not symmetric with respect to positive and negative classes and the real-world datasets we tested have variable skew (Wikipedia: 1460 positives/467 negatives, RTE 400 negatives/positives, Temp 203 negatives/259 positives, Duchenne 101 negatives/58 positives) where the negative class is not always a minority. We thus report the average recall along the datasets in Table 3.1 for all methods.

Though both majority voting and BSP are superior to other methods for the average recall of the Wikipedia dataset, we cannot reject the null hypothesis for the performance difference between them, that is, it is not statistically significant for the size 116
Table 3.1: Comparative average recall for real-world datasets. Statistically significant superiority is marked with bold. We report performance at the fixed point.

<table>
<thead>
<tr>
<th>Method</th>
<th>Wiki</th>
<th>RTE</th>
<th>Temp</th>
<th>Duchenne</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reg. KOS</td>
<td>0.720</td>
<td>0.506</td>
<td>0.534</td>
<td>0.611</td>
</tr>
<tr>
<td>Harmonic</td>
<td>0.782</td>
<td>0.929</td>
<td>0.937</td>
<td>0.731</td>
</tr>
<tr>
<td>BSP</td>
<td><strong>0.810</strong></td>
<td>0.923</td>
<td><strong>0.947</strong></td>
<td><strong>0.752</strong></td>
</tr>
<tr>
<td>Majority</td>
<td><strong>0.810</strong></td>
<td>0.919</td>
<td>0.937</td>
<td>0.743</td>
</tr>
<tr>
<td>EM-AMF</td>
<td>0.762</td>
<td>0.925</td>
<td>0.938</td>
<td>0.468</td>
</tr>
<tr>
<td>EM</td>
<td>0.731</td>
<td>0.925</td>
<td>0.934</td>
<td>0.427</td>
</tr>
</tbody>
</table>

of the samples we use. BSP has superior performance for the NLP Temp and Duchenne experiment datasets, whereas we note that the EM methods obtain a particularly low average recall, even though they emerge as slightly superior for the average error measure which is blind to the classes, which they achieve by labeling items with the label of the majority class in an indiscriminate way.

The reason for the low performance of the iterative methods in some of the real-world datasets is likely to be due to the underlying user model. We use a one-coin model to describe user behavior. In some cases, users do not behave according to the simple Bernoulli assumption. A two-coin model is likely to have benefits on the performance. Also, for the purposes of the study we do not assume any information on the class skew and we have non-informative priors on the items. Our methods can support informative priors. Indeed, modifying the item prior to reflect the class skew provides benefits. Estimating the data skew is feasible in real-world settings by sampling, so we expect our iterative methods to adapt well in a non unsupervised setting where we use existing knowledge on the users and items to impose informative priors.
3.6 Conclusions

We describe two new methods for aggregating binary crowdsourcing tasks. The first, named harmonic, is derived heuristically, and it is simple to describe and implement. The second method has a principled derivation, rooted in the iterative update of estimates of user reliability and item quality distributions via bayesian parameter estimation. Our experimental evaluation indicates that the two algorithms perform better on many occasions, both on synthetic data, and on real-world data. The algorithm based on bayesian parameter estimation exhibits a slightly superior performance. In many practical applications, however, the simplicity, robustness and ease of implementation of the harmonic approach may make it the preferable approach. The practice in the literature is to report results on the fixed point, that is, when iterations stop producing changes. Our experiments, by following performance through iterations, show that the methods can achieve their best performance well before the fixed point, only to occasionally worsen as the fixed point is achieved. We attribute this to the recycling of information after several iterations that induces artificial correlation. We discovered that a good heuristic criterion is to perform \( k = \frac{\log(\max\{m,n\})}{\log(2d)} \) iterations, where \( m, n \) are the numbers of items of users and items, and \( d \) is the geometric average of all node degrees. This helps to propagate all available information across the length of the graph, while keeping correlation low. Stopping after \( k \) iterations instead of converging to the fixed point offers benefits for all iterative methods.
Chapter 4

Learning from Crowdsourced Graphs

Using LSTMs

4.1 Problem setting and contributions

In this chapter, we focus on prediction problems over crowdsourced graphs. Crowdsourcing applications/settings yield irregular graph structures where nodes represent interacting entities, e.g. workers/assignments. The edges of the graphs capture interactions between entities and can contain informative features. Model-based approaches like EM make rigid assumptions on the underlying model of user behavior. Such model-based approaches do not leverage informative features that may be available on the edges. Many prediction problems can be naturally phrased as inference problems over the local neighborhood of a graph. Consider, for instance, crowdsourced grading. We can construct a (bipartite) graph consisting of items and graders, where
edges connect items to users who graded them, and are labeled with the grade assigned (see Figure 4.1). To infer the grade for an item, we can look at the graph involving the adjacent nodes: this graph, known as the 1-neighborhood, consists of the people who graded the item and of the grades they assigned. If we wish to be more sophisticated, and try to determine which of these people are good graders, we could look also at the work performed by these people, expanding our analysis outwards to the 2- or 3-neighborhood of each item.

For another example, consider the problem of predicting which bitcoin addresses will spend their deposited funds in the near future. Though bitcoin user behavior can be deemed as an instance of crowd behavior and therefore it is related to
crowdsourcing, the relation is loose and Bitcoin is not an archetypal example of a crowdsourcing application. However, we include this example as the techniques we propose extend to any irregular graph that contains informative features in its edges and not only graphs from crowdsourcing applications in the strict sense. Bitcoins are held in “addresses”; these addresses can participate in transactions where they send or receive bitcoins. To predict which addresses are likely to spend their bitcoin in the near future, it is natural to build a graph of addresses and transactions, and consider neighborhoods of each address (see Figure 4.2) The neighborhood contains information on where the bitcoins came from, and on what happened to bitcoins at the interacting addresses, which (as we will show) can help predict whether the coins will be transacted soon.

For a third example, consider the problem of predicting user behavior on Wikipedia. Users interact by collaboratively editing articles, and we are interested in predicting which users will have their work reverted. We can build a graph with users as nodes, and interactions as edges: an interaction occurs when two users edit
Figure 4.3: Example of a graph of users interacting over Wikipedia pages. An interesting task is to predict which edit will be reverted in the future based on the user’s interactions in the past.

the same article in short succession, and one either keeps, or undoes, the work of the other. The 1-neighborhood of a user will tell us how often that user’s work has been kept or reverted. (see Figure 4.3) Again, we can consider larger neighborhoods to gather information not only on the user, but on the people the user interacted with, trying to determine whether they are good contributors, how experienced they are, whether they are involved in any disputes, and so forth.

In this chapter, we show how to solve these problems by applying machine learning, using an architecture based on multi-level Long Short-Term Memory (LSTM) neural nets [30, 22, 24], with each LSTM level processing one “degree of separation” in the neighborhood.

The challenge of applying machine learning to graph neighborhoods lies in the fact that many common machine learning methods, from neural nets [32] to support vector machines (SVMs) [10], are set up to handle fixed-length vectors of features as
input. As a graph neighborhood is variable in size and topology, it is necessary to summarize the neighborhood into a fixed number of features to use in learning. Some machine learning methods, such as logistic regression [7], can accept a potentially unbounded number of inputs, but every input has its own index or name, and it is not obvious how to map the local topology of a graph into such fixed naming scheme in a way that preserves the structure, or the useful information.

Machine-learning methods that can learn from sequences, such as LSTMs or recurrent neural nets [66, 29], offer more power. It is possible to traverse the local neighborhood of a node in a graph in some order (pre-, post-, or in-order), and encode the neighborhood in a sequence of features complete with markers to denote edge traversals, and then feed this sequence to an LSTM. We experimented with this approach, but we did not obtain any useful results: the LSTMs were unable to learn anything useful from a flattened presentation of the graph neighborhood.

We propose a learning architecture based on the use of multiple levels of LSTMs. We call our architecture *Multi-Level Sequence Learners* since any structure capable of learning from sequences, and not just LSTMs, can be used. Our architecture performs predictions for one “target” graph node at a time. First, the graph is unfolded from the target node, yielding a tree with the target node as its root at level 0, its neighbors as level-1 children, its neighbors’ neighbors as level-2 children, and so forth, up to a desired depth $D$. At each tree node $v$ of level $0 \leq d < D$, a level-$d+1$ LSTM is fed sequentially the information from the children of $v$ at level $d+1$, and produces as output information for $v$ itself. Thus, we exploit LSTMs’ ability to process sequences of
any length to process trees of any branching factor. The top-level LSTM produces the desired prediction for the target node. The architecture requires training $D$ LSTMs, one per tree level. The LSTMs learn how to summarize the neighborhood up to radius $D$ on the basis of data, avoiding the manual task of synthesizing a fixed set of features. By dedicating one LSTM to each level, we can tailor the learning (and the LSTM size) to the distance from the target node. For instance, in the bipartite graph arising from crowdsourced grading, it is desirable to use different LSTMs for aggregating the edges converging to an item (representing grades received), and for aggregating the edges converting to a user (representing the grades assigned).

A consequence of the local nature of the learning mechanism is that the amount of computation required is independent of the total size of the graph. Indeed, the approach can be applied even when the complete graph is unknown, or too expensive to even construct. In order to train and apply our LSTMs, we simply need a sufficient number of graph neighborhoods to be available for training, testing, and prediction.

We demonstrate the effectiveness of the proposed approach over four problems. The first problem is a synthetic example concerning the crowdsourcing of yes/no labels for items. The other two are based on real data, and they are the previously mentioned problems of aggregating crowdsourced grades and predicting bitcoin spending. In all four problems, we show that the ability of multi-level sequence learners to exploit any feature in the data leads to high performance with minimal feature engineering effort and no apriori model assumptions. We are making available the open-source code implementing LSTMs and multi-level sequence learners, along with the datasets, at
4.2 Related Work

Predicting properties of nodes in graph structures is a common problem that has been widely studied. Several existing approaches view this as a model-based inference problem. A model is created, and its parameters are tuned on the basis of the information available; the model is then used to perform inference. As the exact probabilistic inference is generally intractable [37], most techniques rely on iterative approximation approaches. Iterative approximations are also at the root of expectation maximization (EM) [12]. Iterative parameter estimation has been used, together with Gibbs sampling, to reliably aggregate peer grades in massive on-line courses [47]. Iterative, model-based approaches have also been used for reliably crowdsourcing boolean or multi-class labels [34, 35]. In these works, a bipartite graph of items and workers is created, and then the worker reliabilities, and item labels or grades, are iteratively estimated until convergence.

Compared to these models, the benefit of our proposed approach is that it does not require a model, and thus, it can avail itself of all the features that happen to be available. For instance, in crowdsourced grading, we can use not only the agreement among the graders to judge their reliability, but also any other information that might be available, such as the time taken to grade, or the time of day, or the number of items previously graded by the user, without need to have a model of how these features
might influence grade reliability. We will show that this ability can lead to superior performance compared to EM and [34] when additional features are available. On the other hand, machine-learning based approaches such as ours are dependent on the availability of training data, while model-based approaches can be employed even in its absence.

A work closely related to ours is described in [56], where tree-structured LSTMs are introduced and used to learn distributed representations of sentences of words. The authors present two types of tree-structured LSTMs: child-sum ones, able to deal with arbitrary branching factors, and $N$-ary ones, tailored for a fixed branching factor. In child-sum LSTMs, the memory cell of a node is obtained by summing contributions from its children, each gated according to features of both children and parent. The setting of this work is tailored to NLP tasks; the goal is to improve the prediction of semantic relatedness of two sentences and to classify sentiment. The word representations are initialized using Glove vectors [44], a well-known word embedding technique, and fed as input to the LSTM architecture which produces representations of sentences of words by parsing the syntactic trees of sentences. These embeddings of sentences are in turn used as input to a neural network that performs the prediction task.

The deep convolutional network approach to machine learning has been extended to graphs in [8, 28], where the spectrum of the graph Laplacian is used in lieu of the translations in order to reconstruct a notion of regularity on the graphs. The approach is applied to the Merck Molecular Activity Challenge and to Reuter news datasets, among others. The spectral-based approach requires a consideration of the
Several approaches have been proposed for summarizing graph structures in feature vectors. The algorithm \textit{node2vec} [26] enables the construction of embeddings for graph nodes in such a way that the embedding optimally represents the node’s location in the graph. The objective function models the posterior probability of graph neighborhoods for a given node. The resulting embedding thus summarizes a node’s location in a graph, but it does not summarize the original features of the node, or the possibly existing features of the interactions between a node and neighbors. In contrast, the techniques we introduce allow us to leverage the node and edge features of the graph neighborhood.

In \textit{DeepWalk} [45], feature vectors for graph nodes are constructed by performing random walks from the nodes, and applying various summarization techniques to the list of feature vectors of the visited nodes. This approach enables the consideration of variable-diameter neighborhoods, in contrast to our exploration, which proceeds strictly breath-first. In DeepWalk, nodes that are similar in their features and graph neighborhood are mapped into similar feature vectors. The construction of the summarizing feature vector is guided by considerations of similarity, rather than by backpropagation from the learning goal, as in our approach.

LSTMs were proposed to overcome the problem of vanishing gradient over long sequences that affects recurrent neural nets [30, 22]. LSTMs have been widely useful in a wide variety of learning problems; see, e.g., [25, 55]. Recurrent neural nets and LSTMs have been generalized to multi-dimensional settings [4, 24]. The multi-level architecture
proposed here can handle arbitrary topologies and non-uniform nodes and edges (as in
bipartite graphs), rather than regular n-dimensional lattices, at the cost of exploring
smaller neighborhoods around nodes.

Learning over graphs can be reduced to a standard machine-learning problem
by summarizing the information available at each node in a fixed set of features. This
has been done, for instance, with the goal of link prediction, consisting in predicting
which users in a social network will collaborate or connect next [3]. Graph summariza-
tion typically requires deep insight into the problem, in order to design the summary
features. The multi-level LSTMs we propose here constitute a way of learning such
graph summarization.

Some recent work has looked at the problem of summarizing very large graphs
into feature vectors [57]. The goals (and methods) are thus different from those in the
present chapter, where the emphasis consists in considering nodes together with their
immediate neighborhoods as input to machine learning.

There is much work on learning with graphs, where the graph edges encode the
similarity between the nodes (rather than features, as in our case); see, e.g., [69, 6, 21].

4.3 Learning from Graph Neighborhoods

We consider a graph $G = (V, E)$ with set of vertices $V$ and edges $E \subseteq V \times V$.
We assume that each edge $e \in E$ is labeled with a vector of features $g(e)$ of size $M$.
Each vertex $v \in V$ is associated with a vector of labels. The goal is to learn to predict
the vertex labels on the basis of the structure of the graph and the edge labels.

This setting can model a wide variety of problems. Considering only edge features, rather than also vertex features, involves no loss of generality: if there are interesting features associated with the vertices, they can be included in the edges leading to them. If the goal consists in predicting edge outputs, rather than vertex, one can construct the dual graph $G' = (E, V')$ of $G$, where edges of $G$ are vertices of $G'$, and where $V' = \{(u, v), (v, w)\} | (u, v), (v, w) \in E$.

Learning method overview. Our learning strategy can be summarized as follows. In order to predict the label of a node $v$, we consider the tree $T_v$ rooted at $v$ and with depth $D$, for some fixed $D > 0$, obtained by unfolding the graph $G$ starting from $v$. We then traverse $T_v$ bottom-up, using sequence learners, defined below, to compute a label for each node from the labels of its children edges and nodes in $T_v$. This traversal yields an output label $y_v$ for the root $v$ of the tree. In training, the output $y_v$ can be compared with the desired output, a loss be computed, and backpropagated through the tree. We now present in detail these steps.

Graph unfolding. Given the graph $G = (V, E)$ and a node $v \in V$, along with a depth $D > 0$, we define the full unfolding of $G$ of depth $D$ at $v$ as the tree $T_v$ with root $v$, constructed as follows. The root $v$ has depth 0 in $T_v$. Each node $u$ of depth $k < D$ in $T_v$ has as children in $T_v$ all nodes $z$ with $(u, z) \in E$; the depth of each such $z$ is one plus the depth of $u$. A single graph node may correspond to more than one node in the unfolding. We will rename the nodes of the unfolding so that they are all distinct;
Figure 4.4: An example of a graph and its asymmetric unfolding at node a for depth 2. We rename the nodes that appear in many locations so that they have distinct names, for instance, we use e, e' and e'' to denote the copies of e.

nodes and edges in the unfolding inherit their labels from their correspondents in the graph.

It is possible to perform learning using asymmetric unfolding, in which if a node u has parent u', we let the descendants of u be \{z \mid (u, z) \in E, z \neq u'\}. Figure 4.4 illustrates a graph and its asymmetric tree unfolding at node a and depth 2. Which of the two unfolding is more useful depends on the specifics of the learning problem, and we will discuss this choice in our applications.

**Sequence learners.** Our proposed method for learning on graphs leverages sequence learners. A sequence learner is a machine-learning algorithm that can accept as input an arbitrary-length sequences of feature vectors, producing a single vector as output. *Long Short-Term Memory* neural nets (LSTMs) [30] are an example of such sequence learners. We denote a sequence learner parameterized by a vector \(w\) of parameters by \(L[w]\). In LSTMs, the parameter vector \(w\) consists of the LSTM weights. We say that a sequence learner is of shape \((N, K)\) if it accepts a sequence of vectors of size \(N\), and produces a vector of size \(K\) as output. We assume that a sequence learner \(L[w]\) of shape
$(N, K)$ can perform three operations:

- **Forward propagation.** Given a input sequence $x^{(1)}, x^{(2)}, \ldots, x^{(n)}$, where each $x^{(i)}$ is a vector of size $N$, compute an output $y$, where $y$ is a vector of size $K$.

- **Loss backpropagation.** For a loss function $L$, given $\partial L/\partial y$ for the output, it can compute $\partial L/\partial x^{(j)}$ for each $x^{(1)}, x^{(2)}, \ldots, x^{(N)}$. Here, $\partial L/\partial y$ is a vector having $\partial L/\partial y_i$ as component for each component $y_i$ of $y$, and likewise, $\partial L/\partial x^{(j)}$ is a vector with components $\partial L/\partial x^{(j)}_k$, for each component $x^{(j)}_k$ of $x^{(j)}$.

- **Parameter update.** For a loss function $L$, given $\partial L/\partial y$ for the output, it can compute a vector $\Delta w$ of parameter updates. The parameter updates can be for instance computed via a gradient-descent method, taking $\Delta w = -\alpha \partial L/\partial w$ for some $\alpha > 0$, but the precise method varies according to the structure of the sequence learner; see, e.g., [22].

In an LSTM, backpropagation and parameter update are performed via backpropagation through time; see [64, 66] for details.

### 4.3.1 Multi-Level Sequence Learners

Given a graph $G$ with labeled edges as above, we now describe the learning architecture, and how to perform the forward step of node label prediction, and the backward step of backpropagation and parameter updates. We term our proposed architecture *multi-level sequence learners*, or MLSL, for short.

We start by choosing a fixed depth $D > 0$ for the unfolding. The prediction and
learning is performed via $D$ sequence learners $L_1, L_2, \ldots, L_D$. Each sequence learner $L_i$ will be responsible for aggregating information from children at depth $i$ in the unfolding trees, and computing some information for their parent, at depth $i - 1$. The sequence learner $L_D$ has shape $(M, K_D)$, where $M$ is the size of the edge labels: from the edge labels, it computes a set of features of size $K_D$. For each $0 < d < D$, the sequence learner at depth $d$ has shape $(M + K_{d+1}, K_d)$ for some $K_d > 0$, so that it will be able to aggregate the edge labels and the output of the learners below, into a single vector of size $K_d$.

Note that learners $L_d$ for depth $1 < d \leq D$ can appear multiple times in the tree, once for each node at depth $d - 1$ in the tree. All of these instances of $L_d$ share the same parameters, but are treated separately in forward and backward propagation.

The behavior of these sequence learners is defined by the parameter vectors $w^{(1)}, \ldots, w^{(D)}$; the goal of the learning is to learn the values for these parameter vectors that minimizes the loss function. We stress that the sequence learners $L_1, L_2, \ldots, L_D$ and their parameter vectors $w^{(1)}, \ldots, w^{(D)}$ can depend on the depth in the tree (there are $D$ of them, indeed), but they do not depend on the root node $v$ whose label we are trying to predict.

In order to learn, we repeatedly select root nodes $v^* \in V$, for instance looping over them, or via some probability distribution over nodes, and we construct the unfoldings $T_{v^*}$. We then perform over $T_{v^*}$ the forward and backpropagation steps, and the parameter update, as follows.
**Forward propagation.** The forward propagation step proceeds bottom-up along $T_v$.

Figure 4.5 illustrates how the sequence learners are applied to an unfolding of the root node $a$ of the graph of Figure 4.4 with depth 2 to yield a prediction for node $a$.

- **Depth $D$.** Consider a node $v$ of depth $D - 1$ with children $u_1, \ldots, u_k$ at depth $D$. We use the sequence learner $L_D$ to aggregate the sequence of edge labels $g(v, u_1), \ldots, g(v, u_k)$ into a single label $f(v)$ for $v$.

- **Depth $0 < d < D$.** Consider a node $v$ at depth $d - 1$ with children $u_1, \ldots, u_k$ at depth $d$. We forward to the learner $L_d$ the sequence of vectors $g(v, u_1) \odot f(u_1), \ldots, g(v, u_n) \odot f(u_n)$ obtained by concatenating the feature vectors of the edges from $v$ to the children, with the feature vectors computed by the learners at depth $d + 1$. The learner $L_d$ will produce a feature vector $f(v)$ for $v$.

**Backward propagation.** Once we obtain a vector $y = f(v^*)$ for the root of $T_{v^*}$, we can compute the loss $\mathcal{L}(y)$, and we can compute $\partial \mathcal{L} / \partial y$. This loss is then backpropagated from the root down to the leaves of $T_{v^*}$, following the topology of the tree (refer again to Figure 4.5). Consider a node $v$ at depth $d - 1$, for $0 < d \leq D$, with computed feature vector $f(v)$. We backpropagate through the instance of the learner $L_d$ that computed $f(v)$ the loss, obtaining $\partial \mathcal{L} / \partial x_i$ for the input vectors $x^{(0)}, \ldots, x^{(k)}$ corresponding to the children $u_1, \ldots, u_k$ of $v$.

- If these children are at depth $d < D$, each vector $x^{(j)}$ consists of the concatenation $g(v, u_j) \odot f(u_j)$ of the features $g(v, u_j)$ from the graph edge, and of the features...
Figure 4.5: Forward propagation corresponding to the tree unfolding of Figure 4.4. The elements of the sequence which is fed to learner $L_1$ consist of the features of the respective edges concatenated with the output from learners below. Note the use of three instances of the learner $L_2$, one for each depth-2 node in the unfolding. These instances share the same parameters. In the figure, the symbol $\langle$ denotes the concatenation of feature vectors.

\[
f(u_j)\text{ computed for } u_j. \text{ As the former require no further backpropagation, we retain the portion } \partial L / \partial f(u_j) \text{ for further backpropagation.}
\]

- At the bottom depth $d = D$ of the tree, each vector $x^{(j)}$ corresponds to the graph edge labels $g(v, u_j)$, and backpropagation terminates.

**Parameter update (learning).** Consider a learner $L_d$ for depth $1 \leq d \leq D$, defined by parameters $w^{(d)}$. To update the parameters $w^{(d)}$, we consider all instances $L_d^{(1)}, \ldots, L_d^{(m)}$ of $L_d$ in the tree $T_{v^*}$, corresponding to the nodes $v_1, \ldots, v_m$ at depth $d$ (refer again to Figure 4.5). For each instance $L_d^{(i)}$, for $i = 1, \ldots, m$, from $\partial L / \partial f(v_i)$ we can compute a parameter update $\Delta_i w^{(d)}$. We can then compute the overall parameter update for $L_d$ as the average $\Delta w^{(d)} = (\Delta_1 w^{(d)} + \cdots + \Delta_m w^{(d)}) / m$ of the updates over
the individual instances.

**Preserving learner instance state.** As mentioned above, a sequence learner for a given depth may occur in several instances in the tree obtained by unfolding the graph (see Figure 4.4). Commonly, to perform backpropagation and parameter update though a learner, it is necessary to preserve (or recomputate) the state of the learner after the forward propagation step; this is the case, for instance, both for neural nets and for LSTMs. Thus, even though all learner instances for depth $d$ are defined by a single parameter vector $w^{(d)}$, it is in general necessary to cache (or reconstruct) the state of every learner instance in the tree individually.

### 4.3.2 Training

During training, we repeatedly select a target node, unfold the graph, feed the unfolding to the multi-level LSTMs, obtain a prediction, and backpropagate the loss, updating the LSTMs. An important choice is the order in which, at each tree node, the edges to children nodes are fed to the LSTM. The edges can be fed in random order, shuffling the order for every training sample, or they can be fed in some fixed order. In our applications, we have found each of the two approaches to have uses.

### 4.4 Applications

We have implemented multi-level sequence learners on the basis of an LSTM implementation performing backpropagation-though-time learning [24], which we com-
combined with an AdaDelta choice of learning step [68]. We report the results on one synthetic setting, and three case studies based on real data. The code and the datasets can be found at https://sites.google.com/view/ml-on-structures.

For imbalanced datasets, apart from the accuracy (percentage of correct guesses), we report the average recall, which is the unweighted average of the recall of all classes. This is suitable in the case of classes of different frequencies, since for highly imbalanced datasets it is easy to inflate the accuracy measure by predicting labels of the most frequent classes.

### 4.4.1 Crowdsourcing boolean labels

We considered the common boolean crowdsourcing task where users provide yes/no labels for items. This is modeled as a bipartite graph, with items and users as the two kind of nodes; the edges are labeled with yes/no. The task consists in reconstructing the most likely labels for the items. We generated synthetic data similar to the one used in [34]. In the data, items have a true yes/no label (which is not visible to the inference algorithms), and users have a hidden boolean variable indicating whether they are truthful, or random. Truthful users report the item label, while random users report yes/no with probability 0.5 each. This is also called the *spammer-hammer* user model. We report results for a graph of 3000 users and 3000 items where item labels are balanced (50% yes/ 50% no) and the probability of a user being reliable is 60%. Each item gets 3 votes from different users. We compare three algorithms:

- The iterative algorithm of [34], abbreviated as KOS. The algorithm requires no
prior.

- Expectation Maximization (EM) [12], where user reliability is modeled via a beta distribution. We used an informative prior (shape parameters $\alpha = 1.2$ and $\beta = 1.0$) for the initial beta distribution which reflects the proportion of reliable users in the graph.

- Our multi-level sequence learners with depths 1 and 3, denoted 1-MLSL and 3-MLSL, where the output (and memory) sizes of 3-MLSL are $K_2 = K_3 = 3$. We train on 1,000 items and test on the remaining 2,000.

For multi-level LSTM, we also consider the case where users have an additional observable feature that is correlated to their truthfulness. This represents a feature such as “the user created an account over a week ago”, which is observable, but not part of standard crowdsourcing models. This feature is true for 90% of reliable users and for 40% of unreliable users. We denote the algorithms that have access to this extra feature as 1-LSL+ and 3-LSL+; KOS and EM cannot make use of this feature as it is not part of their model. Our intent is to show how machine-learning approaches such as MLSLs can increase their performance by considering additional features, independently of a model.

We report the results in Table 4.1. When no additional information is available, EM is superior to 1-MLSL and slightly superior to 3-MLSL. When the additional feature is available, both 1-MLSL+ and 3-MLSL+ learn its usefulness, and perform best.
<table>
<thead>
<tr>
<th>Method</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>KOS</td>
<td>0.8016</td>
</tr>
<tr>
<td>EM</td>
<td>0.9136</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-MLSL</td>
<td>0.8945</td>
</tr>
<tr>
<td>3-MLSL</td>
<td>0.9045</td>
</tr>
<tr>
<td>1-MLSL+</td>
<td>0.9565</td>
</tr>
<tr>
<td>3-MLSL+</td>
<td>0.9650</td>
</tr>
</tbody>
</table>

Table 4.1: Performance of KOS [34], EM (Expectation Maximization) and multi-level sequence learners (MLSLs) of different depths.

4.4.2 Peer Grading

We considered a dataset containing peer grading data from computer science classes. The data comes from an online tool that lets students submit homework and grade each other’s submissions. Each submission is typically reviewed by 3 to 6 other students. The data is a bipartite graph of users and submissions, as in the previous crowdsourcing application. Users assign grades to items in a predefined range (in our case, all grades are normalized in the 0-10 range). Each edge is labeled with the grade, and with some additional features: the time when the student started grading the submission, and the time when they submitted the grade. We treat this as a classification task, where the classes are the integer grades 0, 1, . . . , 10; the ground truth is provided by instructor grades, available on a subset of submissions. Our dataset contained 1,773 labeled (instructor-graded) submissions; we used 1,500 for training and 273 for testing.

We compare three methods. One is simple average of provided grades, rounded to the closest integer. Another method is based on expectation maximization (EM), iteratively learning the accuracy of users and estimating the grades. Finally, we employed
<table>
<thead>
<tr>
<th>Method</th>
<th>Accuracy</th>
<th>Average Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>0.5432</td>
<td>0.3316</td>
</tr>
<tr>
<td>EM-based</td>
<td>0.5662</td>
<td>0.3591</td>
</tr>
<tr>
<td>1-MLSL</td>
<td>0.6044</td>
<td>0.3897</td>
</tr>
<tr>
<td>2-MLSL</td>
<td>0.6010</td>
<td>0.3913</td>
</tr>
</tbody>
</table>

Table 4.2: Performance of EM and 1,2-depth MLSL on peer grading data.

MLSL with the following features (derived from the graph): the time to complete a review, the amount of time between review completion and review deadline, and the median grade received by the student in the assignment. The output of the learner at level 2 is of size 3 where it reaches its peak for this experiment.

Table 4.2 shows the results. The 1- and 2-depth MLSL methods are superior to both the EM-based approach and average. Average recall appears low due to the very high class imbalance of the dataset: some low homework grades are very rare, and mistakes in these rare grades have high impact.

4.4.3 Prediction of Bitcoin Spending

The blockchain is the public immutable distributed ledger where Bitcoin transactions are recorded [43]. In Bitcoin, coins are held by addresses, which are hash values; these address identifiers are used by their owners to anonymously hold bitcoins, with ownership provable with public key cryptography. A Bitcoin transaction involves a set of source addresses, and a set of destination addresses: all coins in the source addresses are gathered, and they are then sent in various amounts to the destination addresses.

Mining data on the blockchain is challenging [40] due to the anonymity of
addresses. We use data from the blockchain to predict whether an address will spend the funds that were deposited to it.

We obtain a dataset of addresses by using a slice of the blockchain. In particular, we consider all the addresses where deposits happened in a short range of 101 blocks, from 200,000 to 200,100 (included). They contain 15,709 unique addresses where deposits took place. Looking at the state of the blockchain after 50,000 blocks (which corresponds to roughly one year later as each block is mined on average every 10 minutes), 3,717 of those addresses still had funds sitting: we call these “hoarding addresses”. The goal is to predict which addresses are hoarding addresses, and which spent the funds. We randomly split the 15,709 addresses into a training set of 10,000 and a validation set of 5,709 addresses.

We built a graph with addresses as nodes, and transactions as edges. Each edge was labeled with features of the transaction: its time, amount of funds transmitted, number of recipients, and so forth, for a total of 9 features. We compared two different algorithms:

- Baseline: an informative guess; it guesses a label with a probability equal to its

<table>
<thead>
<tr>
<th></th>
<th>Accuracy</th>
<th>Avg. Recall</th>
<th>F-1 'spent'</th>
<th>F-1 'hoard'</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>0.6325</td>
<td>0.4944</td>
<td>0.7586</td>
<td>0.2303</td>
</tr>
<tr>
<td>1-MLSL</td>
<td>0.7533</td>
<td>0.7881</td>
<td>0.8172</td>
<td>0.6206</td>
</tr>
<tr>
<td>2-MLSL</td>
<td>0.7826</td>
<td>0.7901</td>
<td>0.8450</td>
<td>0.6361</td>
</tr>
<tr>
<td>3-MLSL</td>
<td>0.7731</td>
<td>0.7837</td>
<td>0.8367</td>
<td>0.6284</td>
</tr>
</tbody>
</table>

Table 4.3: The prediction results on blockchain addresses using baseline approach, and MLSL of depths 1, 2, 3.
percentage in the training set.

- MLSL of depths 1, 2, 3. The outputs and memory sizes of the learners for the reported results are $K_2 = K_3 = 3$. Increasing these to 5 maintained virtually the same performance while increasing training time. Using only 1 output and memory cell was not providing any advances in performance.

Table 4.3 shows the results. Using the baseline we get poor results; the F-1 score for the smaller class (the ‘hoarding’ addresses) is particularly low. Tapping the transaction history and using only one level the learner already provides a good prediction and an average recall approaching 80%. Increasing the number of levels from 1 to 2 enhances the quality of the prediction as it digests more information from the history of transactions. Increasing the levels beyond 2 does not lead to better results, with this dataset.

4.4.4 Discussion

The results from the above applications show that MLSL can provide good predictive performance over a wide variety of problems, without need for devising application-tailored models. If sufficient training data is available, MLSL can use the graph representation of the problem and any available features to achieve high performance.

One of our conclusions is that the order of processing the nodes during training matters. In crowdsourced grading, randomly shuffling the order of edges for a learning
instance as it is used in different iterations during the training process, was superior to using a fixed order. For Bitcoin, on the other hand, feeding edges in temporal order worked best. This seems intuitive, as the transactions happened in some temporal order.

One challenge was the choice of learning rates for the various levels. As the gradient backpropagates across the multiple levels of LSTMs, it becomes progressively smaller. To successfully learn we needed to use different learning rates for the LSTMs at different levels, as the top levels will tend to learn faster.
Bibliography


*Journal of the Royal Statistical Society. Series B (Methodological)*, pages 262–268, 
1977.

[16] Cynthia Dwork, Ravi Kumar, Moni Naor, and D. Sivakumar. Rank aggregation 

Comparing and aggregating rankings with ties. In *Proceedings of the twenty-third 
ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 

[18] Ronald Fagin, Ravi Kumar, and D. Sivakumar. Efficient similarity search and 
classification via rank aggregation. SIGMOD ’03, pages 301–312, New York, NY, 
USA, 2003. ACM.

[19] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms 
for middleware. In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART 
symposium on Principles of database systems*, PODS ’01, pages 102–113, New York, 
NY, USA, 2001. ACM.


[21] Eyal En Gad, Akshay Gadde, A. Salman Avestimehr, and Antonio Ortega. Ac-
tive learning on weighted graphs using adaptive and non-adaptive approaches. In


[28] Mikael Henaff, Joan Bruna, and Yann LeCun. Deep Convolutional Networks on

[29] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, and Jrgen Schmidhuber. Gradient
flow in recurrent nets: the difficulty of learning long-term dependencies. A field


[31] Lu Hong, Scott E Page, and Maria Riolo. Incentives, information, and emergent


[33] Panos Ipeirotis. Mechanical turk: Now with 40.92% spam. Behind Enemy Lines
blog, 2010.

[34] David R. Karger, Sewoong Oh, and Devavrat Shah. Iterative learning for reliable


[51] Aashish Sheshadri and Matthew Lease. Square: A benchmark for research on


[66] Ronald J. Williams and David Zipser. Gradient-based learning algorithms for re-

