Cost and Quality Trade-Offs in Crowdsourcing

Anja Gruenheid
Systems Group
ETH Zurich
anja.gruenheid@inf.ethz.ch

Donald Kossmann
Systems Group
ETH Zurich
donaldk@inf.ethz.ch

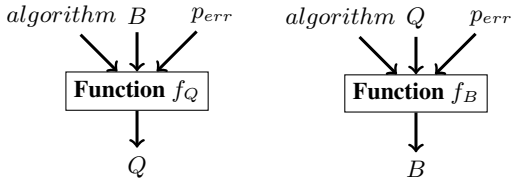
ABSTRACT

Algorithms for crowdsourced tasks such as entity resolution, sorting, etc. have been subject to a variety of research work. So far, all of this work has focused on one specific problem respectively. In this paper, we want to focus on the bigger picture. More specifically, we want to show how it is possible to estimate the budget or the quality of an algorithm in a crowdsourcing environment where noise is introduced through incorrect answers by crowd workers. Such estimates are complex as noise in the information set changes the behavior of established algorithms. Using two sorting algorithms, QuickSort and BubbleSort as examples, we will illustrate how algorithms handle noise, which measures can be taken to make them more robust, and how these changes to the algorithms modify the budget and quality estimates of the respective algorithm. Finally, we will present an initial idea of how such an estimation framework may look like.

1. INTRODUCTION

Quality and its influence on the output result of any algorithm using crowdsourcing has been subject to a range of research work. Research focused on topics such as increasing the worker's motivation, quality control mechanisms, or accepting that workers make mistakes and constructing fault-tolerant variations of the original algorithms [4, 15]. This work will focus on none of these quality assurance methods specifically but on assessing the inter-relationships of the crowdsourcing budget, the crowd worker error rate, and the result quality. Our goal is to define an intuition on how these parameters are interleaved, meaning a) how for example adding votes changes the result quality or b) if a certain result quality is required, how many votes are required to meet these quality constraints. These two scenarios are visualized in Figure 1. Obviously, functions f_Q and f_B are closely related for the sake of conformity, where $f_Q(B_i, p_{err}) = Q_i$ and $f_B(Q_i, p_{err}) = B_i$ holds for a specific input budget B_i and input quality Q_i. In other words, f_Q is the reverse calculation of f_B. Note that the crowd worker error rate is a given parameter in both scenarios that obviously influences the result quality and budget requirements.

It is not the objective of this work to define how this error rate can be determined but we assume that it is a known parameter which can be estimated through worker quality tests or more advanced quality techniques [5, 9]. In addition to the crowd worker error rate and the monetary budget (resp. the required quality), the estimation function takes as input an algorithm which interprets the feedback of the crowd. Depending on this algorithm and its efficiency in a noisy environment, the estimated budget or quality may vary. For the purpose of this paper, we will assume that the algorithm that we want to estimate is a sorting algorithm but the quality and budget estimation techniques that we will present here can also be used for other algorithms in which the crowd is employed, such as entity resolution.



(a) Quality based on Budget (b) Budget based on Quality

Figure 1: Estimation Functions

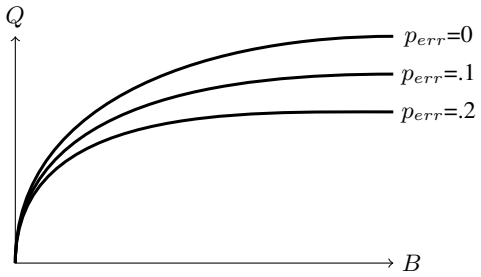


Figure 2: Cost-Quality Trade-Off

Independent of the algorithm, we assume that the quality of the result decreases if the noise in the crowd answers increases which is depicted in Figure 2. We will observe this behavior when examining two example sorting algorithms, QuickSort and BubbleSort [11]. For these algorithms, we will discuss quality and budget trade-offs, showing how fault-tolerance mechanisms and specific adaptations to the noisy environment can improve result quality effectively. Additionally, we will show that the result quality and budget requirements also highly depend on the applied algorithm and we present examples where algorithms that were thought to be better alternatives will produce results with lower quality.

More specifically, we will first describe, using QuickSort and

BubbleSort as example algorithms, how different algorithms handle noise with and without additional fault-tolerance mechanisms and how they compare in terms of the amount of the budget that they use and the quality of their results. From this specific example, we will then draft the challenges of any estimation framework in this kind of environment.

2. RELATED WORK

Crowdsourcing in context of database systems has been subject to a variety of research recently. One research area focuses on integrating crowdsourcing functionality efficiently into database management systems while another research focus are algorithms suitable for this kind of environment. Systems such as CrowdDB [1], Qurk [7], and Deco [10] connect traditional data storage systems with crowdsourcing platforms such as Amazon Mechanical Turk to gain additional information on queries [12]. To enable users to have similar functionality in those systems as in any comparable relational database management system, research has further focused on optimizing crowd accesses (i.e. reducing the budget spent on the crowd) when implementing algorithms such as ranking and entity resolution. For these algorithms, research has pursued two different directions: The first class of algorithms focused on reducing the budget while assuming that the crowd answers perfectly [13, 14], observing the quality of the answers by the crowd workers as an orthogonal problem. The second class of algorithms introduces the notion of fault-tolerance to be able to handle noisy answers from the crowd [2, 3, 8, 6].

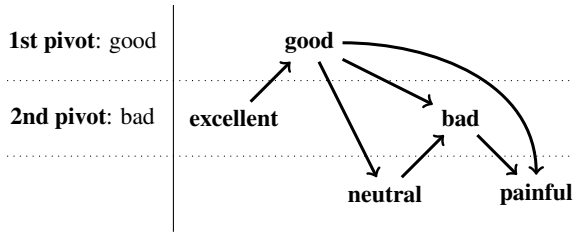
The novelty of our work is that we abstract from the actual implementation of the algorithms in that we want to describe a more general framework for the interdependencies between quality and budget in a crowdsourcing environment independent of the applied algorithm. Hence, we observe rather than construct the behavior of a set of algorithms and show through examples which behavior fits better for noisy environments.

3. IMPACT OF NOISE

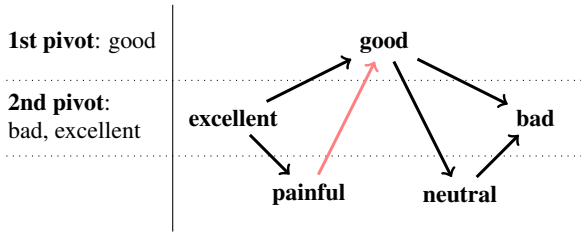
Traditionally, sorting algorithms such as QuickSort and BubbleSort have been evaluated with respect to time and space requirements. The crowdsourcing context now adds another dimension, namely result quality, as a noisy environment influences the quality and correctness of the output of these algorithms negatively. Moreover, temporal constraints become neglible as crowdsourcing itself does not necessarily need immediate response algorithms by design while space requirements can be seen as budget constraints (i.e. the number of comparisons that are required to find a solution). We will show in the following that the choice of algorithm for a certain problem heavily influences the result quality especially in noisy environments. To that purpose, we will first evaluate QuickSort in a crowdsourcing environment after which we will focus on BubbleSort as example algorithms.

3.1 QuickSort

The prerogative of QuickSort is that it can efficiently sort an input set, using $O(n\ log(n))$ comparisons on average. Thus, it is a prime candidate for sorting in a crowdsourcing environment as it allows to reduce the budget spent on the task. On the other hand, noise in the crowdsourcing answers directly propagates if QuickSort is used. An example for this observation is shown in Figure 3. Imagine a crowdsourcing task which requires the workers to order a few words according to their positivity. In the first example answer set, none of the edges (i.e. the votes of crowd workers) is wrong which leads to low budget result with perfect quality. In contrast, in the second



(a) QuickSort Without Noise



(b) QuickSort With Noise

Figure 3: QuickSort Behavior Variations

example set the edge between 'painful' and 'good' distorts the result in that painful is now ranked higher than three other words due to the transitive propagation of this specific wrong comparison.

How noise affects the output of the QuickSort algorithm is dominated by the distribution of the noisy answers and whether the implementation of QuickSort used in this context is fault-tolerant. Many of the current crowdsourcing solutions do not provide fault-tolerant algorithms but assume the crowd to answer perfectly [13, 14]. In contrast, we believe that given the application of these algorithms where information is collected in crowdsourcing platforms such as Amazon Mechanical Turk, accepting that the crowd does create a certain amount of noise is only natural. Mechanisms that can make QuickSort fault-tolerant may include strategies such as using the majority of the crowd votes in order to determine an edge or to request that one answer exceeds the alternative by quorum votes. Depending on the strategy, the estimates of functions f_Q and f_B vary. For example imagine that p_{err}=.2, meaning here that 20% of the actual comparisons are wrong on average. If the majority strategy is applied and all false votes are countered by more true votes, the quality of the result can still be impeccable. On the other hand, if no fault-tolerant mechanism is used, the result quality will suffer because wrong comparisons will be propagated directly into the result. Obviously, this is the best case scenario for the majority strategy and in practice even with fault-tolerance comparisons will be propagated wrongly into the decision structure. But given the integrated quality assurance mechanism of the majority strategy, a smaller number of comparisons will be wrong which means higher result quality in the end. Thus, our first observation is that depending on the interpretation strategy used (for example the *direct* propagation of votes or the *majority* strategy explained in the previous example), the budget and the quality output changes.

Figure 4 depicts a simulated budget-quality trade-off for the two discussed strategies, the direct propagation of votes and the majority interpretation strategy. It shows that fault-tolerance increases the budget requirements for QuickSort but it also improves the result quality, i.e. the number of pair-to-pair comparisons that are confirmed by the ground truth. To better understand this observation take again the running example: If the crowd worker error rate is

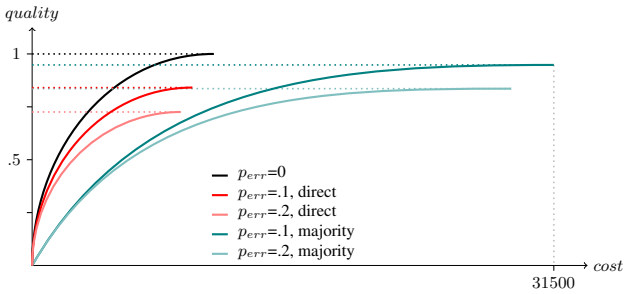


Figure 4: Fault-Tolerance Strategy Trade-Offs for QuickSort

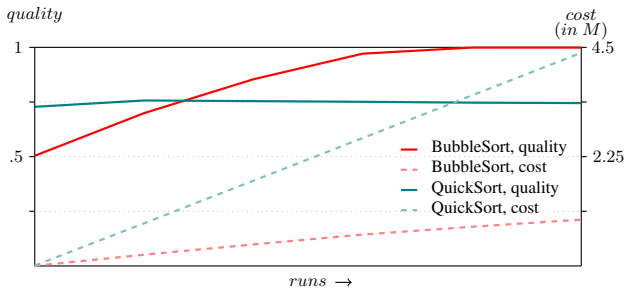


Figure 5: QuickSort vs BubbleSort, $p_{err} = .2$

p_{err}=.2, at least one edge is faulty out of the six that we need for our QuickSort solution in Figure 3. The result quality of our example is improved if the majority interpretation strategy is applied because on average, the erroneous votes are better distributed than in the direct scenario thus resulting in less erroneous comparisons as exemplified above. The figure also visualizes that applying the majority strategy leads to a higher budget as a comparison is made multiple times, thus the overall information gain is slower per unit of budget.

During the simulations a decrease in cost can be observed when the error rate increases, i.e. the cost for QuickSort with the direct strategy and without noise is approximately 11,000 while the cost decreases for $p_{err} = .1$ to 9,700. The explanation for this behavior is that in cases where the pivot element is not a central element in the final sorting, wrong answers may benefit the overall number of comparisons as the wrongly judged item is transfered into a smaller bucket and thus used less often overall for comparisons.

Overall, the main observation for QuickSort here is that if the error increases, fault-tolerance measurements can help to improve the quality of the result set even if they also mean an increase in budget in order to reach the break-even point. On the other hand, it is essential to observe that even with fault-tolerant techniques, there is no guarantee that QuickSort will provide a correct result to the problem of sorting an input dataset. Thus, if the problem statement is to find a complete ranking of the input set, QuickSort fails as a sorting algorithm in this example setting.

3.2 BubbleSort

In contrast to QuickSort, BubbleSort is considered to be suboptimal in terms of the number of comparisons it needs to find a solution as it does local comparisons rather than global comparisons and its average comparison complexity is $O(n^2)$. In a noisy environment predicates shift and the number of pure item comparisons is not the decisive factor anymore as we will show in the following. BubbleSort specifically has one characteristic that makes this algorithm suitable for noisy environments: It benefits from the **input order** of the records. Thus, if the input is already sorted, the algorithm will run in $O(n)$. This characteristic can be leveraged by running BubbleSort multiple times over the same input dataset. Due to the input sensitivity of BubbleSort, subsequent *runs* will have a decreased number of comparisons. BubbleSort in a noisy environment will further benefit from a very intuitive modification to the algorithm that is made due to the cost sensitivity of the environment: Every comparison that is requested is stored. As a result, comparing the same item twice will not result in additional cost but if a crowd worker returns the wrong answer this answer will be propagated through the dataset. Even though this kind of propagation reduces the quality of the output, the cost of iteratively improving the sorting solution is less than running BubbleSort without storage mechanisms as verified through simulations.

These modifications are minor and do not change the character of the algorithm which proceeds by doing local comparisons and 'bubbling' items to their correct spot in the ordering.

3.3 QuickSort vs. BubbleSort

When comparing QuickSort and BubbleSort and observing their behavior in a noisy setting, we can see that traditional algorithmic assumptions shift. Consider Figure 5 which shows an example sketch for this shift: Here, we draw the behavior of the direct QuickSort approach and the BubbleSort variation described above in terms of cost and quality simulated in a noisy environment with $p_{err} = .2$ and 1000 items to be sorted. As we want to obtain the optimal result quality eventually, we execute both algorithms multiple times, the number of runs in this simulation is 500. There are two main observations that can be drawn from these simulations.

First, they show that BubbleSort can improve its output result over time due to its input sensitivity while the output quality of QuickSort remains constant. Additionally, the average number of crowd comparisons requested in BubbleSort decreases over time (in our simulations up to 50%). Thus, the output quality of BubbleSort increases because if the algorithm asks for fewer input from the crowd, fewer noise is introduced into the sorting. In contrast, the level of noise that the QuickSort algorithm has to handle remains the same over multiple runs. The second observation that can be drawn from these simulations is that even if cost reduction techniques are used, BubbleSort is in this setup 26 times more expensive than QuickSort when achieving the same quality (i.e. at the quality break-even point). Thus, even though the increase in cost is lower, the necessity of repeating the algorithm multiple times increases the overall cost for BubbleSort in the end.

The example above shows that some algorithms like BubbleSort have a natural robustness that makes them more suitable in noisy environments. Thus, we think that established algorithms that outperform others according to traditional complexity theory need to be looked upon from a new angle in noisy environments as an alternative or as a component in hybrid setups.

4. QUALITY & BUDGET ESTIMATES

In order to realize an estimation framework, we propose as first step to establish a modified complexity theory that is noise-aware. That is, in traditional complexity theory the number of comparisons per algorithm determines the usefulness of that specific algorithm. As we have shown previously, traditional assumptions may not hold in the crowdsourcing environment due to noise in the information set. Thus, algorithms need to be evaluated taking into consideration their behavior in noisy environments. This evaluation can then be used to determine which algorithm is most suitable to solve a certain problem. As a result, we propose to define a new model for the

complexity of an algorithm that is dependent on a) the number of records in the input set (analogous to traditional complexity theory) b) and the amount of noise that this algorithm will encounter.

If the complexity of an algorithm is known, we can then use it to form estimates of the required budget given certain quality constraints or estimates that approximate the result quality given certain budget constraints for an algorithm as shown initially in Figure 1. Note that this notion of complexity is not restricted to sorting algorithms but it can be used for evaluating a variety of algorithms such as entity resolution, finding the maximum etc. To be able to take input parameters such as a required quality or available budget, traditional algorithms need to be adjusted. Modifying these algorithms is necessary because current algorithms take an input dataset after which they are executed to return a result with a certain quality for a certain budget independent of the crowd worker error rate. How budget and quality are distributed is subject to the specific execution run and resemble points on a curve similar to those depicted in Figure 5 where the curve varies according to the level of noise in the information set and the chosen algorithm.

In our opinion, algorithms need to be conscious of input constraints to be able to compare two algorithms in a noisy environment. Going back to our comparison of QuickSort and BubbleSort, we can determine that according to the characteristics of the algorithms, QuickSort with fault-tolerance can result in good quality results in low noise environments while BubbleSort can be more adequate in environments where the crowd answers wrongly more often. But in order to be able to define the actual trade-off, we need to compare how the algorithms react in scenarios with fixed error rate and budget or quality. Thus, we want to be able to define the cost-quality function curves instead of points on the curve that can be derived from current algorithm variations that are not constraint-aware as mentioned previously.

If these curves are known, we can use them for our estimation framework in that they define the complexity of an algorithm dependent on the crowd worker error rate where we can vary budget and quality constraints in order to find a suitable estimated output. A simple two-step estimation process may then look as follows:

1. **Initial Estimate** - Given the properties of the chosen algorithm, i.e. its complexity in this kind of noisy environment, decide upon an initial estimate.

2. **Adjust Estimate** - Using the knowledge of the input quality respectively budget parameter, adjust the estimate as to meet/exhaust the parameter. An example measure is to allocate more budget for multiple runs of the same algorithm or for fault-tolerant mechanisms.

Obviously, this process is only a rough sketch of the functionality of such an estimation framework. More specific implementation and design details of these two steps are subject to future work as is the specification of the complexity theory that enables us to give accurate estimates in noisy environments.

5. CONCLUSION

In this work, we have presented several observations made for quality and cost trade-offs in the still novel crowdsourcing environment. Using QuickSort and BubbleSort as example algorithms, we have shown that any estimation function that wants to determine the expected output of that algorithm has to take into consideration that different algorithms behave differently when confronted with noise. Their behavior not only changes in terms of budget requirements but they also vary in their robustness in noisy environments. We have presented techniques such as fault-tolerant interpretation strategies and increasing the number of executions of the algorithms that influence the result estimation and choice of algorithm when taken into consideration. Last, we have shown that traditional beliefs such that QuickSort is better than BubbleSort due to its lower number of comparisons are questioned in a noisy environment because even though QuickSort is requires a smaller budget, BubbleSort may provide better quality results due to its ingrained second chance mechanism and its ability to leverage information over multiple runs.

In the future, we plan to specify our sketched estimation framework and to define exactly how noise can be evaluated in such an environment. Additionally, we want to determine whether it is possible to formally analyze the trade-off between cost and quality for algorithms that are traditionally integrated into relational database management systems. Knowing the complexity of these algorithms will help us to determine which of them are viable options for a database system that incorporates crowdsourcing as a way to obtain information.

6. REFERENCES

[1] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. Crowddb: answering queries with crowdsourcing. In *SIGMOD Conference*, pages 61–72, 2011.

[2] A. Gruenheid, D. Kossmann, S. Ramesh, and F. Widmer. Crowdsourcing entity resolution: When is a=b? Technical report, ETH Zurich, 2012.

[3] S. Guo, A. Parameswaran, and H. Garcia-Molina. So Who Won? Dynamic Max Discovery with the Crowd. In *Proceedings SIGMOD*, 2012. to appear.

[4] A. Kittur, J. V. Nickerson, M. Bernstein, E. Gerber, A. Shaw, J. Zimmerman, M. Lease, and J. Horton. The future of crowd work. In *Proceedings of the 2013 conference on Computer supported cooperative work*, CSCW '13, pages 1301–1318, New York, NY, USA, 2013. ACM.

[5] M. Lease. On quality control and machine learning in crowdsourcing. In *Human Computation*, 2011.

[6] A. Marcus, D. Karger, S. Madden, R. Miller, and S. Oh. Counting with the crowd. In *Proceedings of the 39th international conference on Very Large Data Bases*, PVLDB'13, pages 109–120. VLDB Endowment, 2013.

[7] A. Marcus, E. Wu, D. R. Karger, S. Madden, and R. C. Miller. Demonstration of qurk: a query processor for humanoperators. In *SIGMOD Conference*, pages 1315–1318, 2011.

[8] A. Marcus, E. Wu, D. R. Karger, S. Madden, and R. C. Miller. Human-powered sorts and joins. *PVLDB*, 5(1):13–24, 2011.

[9] D. Oleson, A. Sorokin, G. P. Laughlin, V. Hester, J. Le, and L. Biewald. Programmatic gold: Targeted and scalable quality assurance in crowdsourcing. In *Human Computation*, 2011.

[10] A. Parameswaran, H. Park, H. Garcia-Molina, N. Polyzotis, and J. Widom. Deco: Declarative crowdsourcing. Infolab Technical Report, Stanford University, November 2011.

[11] R. Sedgewick and K. Wayne. *Algorithms, 4th Edition*. Addison-Wesley, 2011.

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

[13] J. Wang, T. Kraska, M. J. Franklin, and J. Feng. Crowder: Crowdsourcing entity resolution. *PVLDB*, 5(11):1483–1494, 2012.

[14] S. E. Whang, P. Lofgren, and H. Garcia-Molina. Question selection for crowd entity resolution. Technical report, Stanford University.

[15] X. Wu, W. Fan, and Y. Yu. Sembler: Ensembling crowd sequential labeling for improved quality. In *AAAI*, 2012.