Argonaut: Macrotask Crowdsourcing for Complex Data Processing

Daniel Haas[◇] Jason Ansel[♣] Lydia Gu[♣] Adam Marcus[♡]

AMPLab, UC Berkeley[◇] GoDaddy, Inc.[♣] Unlimited Labs[♡]
dhaas@cs.berkeley.edu {jansel, lgu}@godaddy.com marcua@marcua.net

ABSTRACT

Crowdsourced workflows are used in research and industry to solve a variety of tasks. The databases community has used crowd workers in query operators/optimization and for tasks such as entity resolution. Such research utilizes *microtasks* where crowd workers are asked to answer simple yes/no or multiple choice questions with little training. Typically, microtasks are used with voting algorithms to combine redundant responses from multiple crowd workers to achieve result quality. Microtasks are powerful, but fail in cases where larger context (e.g., domain knowledge) or significant time investment is needed to solve a problem, for example in large-document structured data extraction.

In this paper, we consider context-heavy data processing tasks that may require many hours of work, and refer to such tasks as *macrotasks*. Leveraging the infrastructure and worker pools of existing crowdsourcing platforms, we automate macrotask scheduling, evaluation, and pay scales. A key challenge in macrotask-powered work, however, is evaluating the quality of a worker's output, since ground truth is seldom available and redundancy-based quality control schemes are impractical. We present Argonaut, a framework that improves macrotask powered work quality using a hierarchical review. Argonaut uses a predictive model of worker quality to select trusted workers to perform review, and a separate predictive model of task quality to decide which tasks to review. Finally, Argonaut can identify the ideal trade-off between a single phase of review and multiple phases of review given a constrained review budget in order to maximize overall output quality. We evaluate an industrial use of Argonaut to power a structured data extraction pipeline that has utilized over half a million hours of crowd worker input to complete millions of macrotasks. We show that Argonaut can capture up to 118% more errors than random spot-check reviews in review budget-constrained environments with up to two review layers.

Articles from this volume were invited to present their results at the 41st International Conference on Very Large Data Bases, August 31st - September 4th 2015, Kohala Coast, Hawaii.
Proceedings of the VLDB Endowment, Vol. 8, No. 12
Copyright 2015 VLDB Endowment 2150-8097/15/08.

1. INTRODUCTION

Crowdsourcing has recently been used to improve the state of the art in areas of data processing such as entity resolution, structured data extraction, and data cleaning. Human computation is commonly used for both processing raw data and verifying the output of automated algorithms. An important concern when assigning work to crowd workers with varying levels of ability and experience is maintaining high-quality work output. Thus, a prominent focus of the crowdsourcing literature has been on quality control: developing workflows and algorithms to reduce errors introduced by workers either unintentionally (due to innocent mistakes) or maliciously (due to collusion or spamming).

Most research on quality control in crowdsourced workflows has focused on platforms that define work as *microtasks*, where workers are asked simple questions that require little context or training to answer. Microtasks are an attractive unit of work, as their small size and low cost make them amenable to quality control by assigning a task to multiple workers and using worker agreement or voting algorithms to surface the correct answer. For example, a common microtask is image annotation, where crowd workers help label an object in an image. As more and more workers agree on an annotation, the confidence of that annotation increases. Microtask research has focused on different ways of controlling this voting process while identifying the reliability of workers through their participation.

Unfortunately, not all types of work can be effectively decomposed into microtasks. Tasks that require global context (e.g., creating papers or presentations) are challenging to programmatically sub-divide into small units. Additionally, voting strategies as a method of quality control break down when applied to tasks with complex outputs, because it is unclear how to perform semantic comparisons between larger and more free-form results. An alternative to seeking out good workers on microtask platforms and decomposing their assignments into microtasks is to recruit crowd workers to perform larger and more broadly defined tasks over a longer time horizon. Such a model allows for in-depth training, arbitrarily long-running tasks, and flexible compensation schemes. There has been little work investigating quality control in this setting, as the length, difficulty, and type of work can be highly variable, and defining metrics for quality can be challenging.

In this paper, we use the term *macrotask* to refer to such complex work. Macrotasks represent a trade-off between microtasks and freelance knowledge work, in that they provide the automation and scale of microtasks, while enabling much of the complexity of traditional knowledge work. We discuss both the limitations and the opportunities provided by macrotask processing, and then present Argonaut, a framework that extends existing data processing systems with the ability to use high-quality crowdsourced macrotasks. Argonaut presents the output of automated data processing techniques as the input to macrotasks and instructs crowd workers to eliminate errors. As a result, it easily extends existing automated systems with human workers without requiring the design of custom-decomposed microtasks.

Argonaut leverages several cost-aware techniques for improving the quality of worker output. These techniques are domain-independent, in that they can be used for any data processing task and crowd work platform that collects and maintains basic data on individual workers and their work history. First, Argonaut organizes the crowd hierarchically to enable trusted workers to review, correct, and improve the output of less experienced workers. Second, Argonaut provides a predictive model of task error, called the TaskGrader, to effectively allocate trusted reviewers to the tasks that need the most correction. Third, Argonaut tracks worker quality over time in order to promote the most qualified workers to the top of the hierarchy. Finally, given a fixed review budget, Argonaut decides whether to allocate reviewer attention to an initial review phase of a task or to a secondary review of previously reviewed tasks in order to maximize overall output quality. We provide an evaluation of these techniques on a production structured data extraction system used in industry at scale. For review budget-constrained workflows, we show up to 118% improvement over random spot checks when combining TaskGrader with a two-layer review hierarchy, with greater benefits at more constrained budgets.

In summary, this paper makes the following contributions:

1. Argonaut, a framework for managing macrotask-based workflows and improving their output quality given a fixed budget and fixed throughput requirement.
2. A hierarchical review structure that allows expert workers to catch errors and provide feedback to entry-level workers on complex tasks. Argonaut models workers and promotes the ones that efficiently produce the highest-quality work to reviewer status. We show that 71.8% of tasks with changes from reviewers are improved.
3. A predictive model of task quality that selects tasks likely to have more error for review. Experiments show that generalizable features are more predictive of errors than domain specific ones, suggesting that Argonaut's models can be implemented in other settings with little task type specific instrumentation.
4. Empirical results that show that under a constrained budget where not every task can be reviewed multiple times, there exists an optimal trade-off between one-level and two-level review that catches up to 118% more errors than random spot checks.

2. RELATED WORK

There is a rich body of research around improving work quality for microtasks. Prior research has described techniques for weighing worker responses based on quality inferred from previous answers to categorical microtasks (e.g., yes/no questions) [13, 15]. Bernstein et. al. present the Find-Fix-Verify design pattern [6], in which workers vote on the responses of other workers to identify good results. Similar to our approach, Rzeszotarski et al. [27] train a model with worker behavioral information (e.g., scrolling, mouse movements, completion time) to classify suspect responses with 80% accuracy. This technique is complementary to our use of worker history and behavioral information, and we extend these approaches to the macrotask context.

Certain types of work, such as entity resolution, has been explored in depth. Work on entity matching has developed good blocking rules for reducing the number of pairs to be matched [9] and found efficient matching algorithms to decide whether a pair of entities are equivalent [3, 5]. Crowdsourcing is frequently used to verify predicted matches [31, 10, 12] and provide training data for matching algorithms [22, 32, 12]. Data Tamer [29] uses a crowd of domain experts to resolve uncertain examples in schema integration, data cleaning and entity resolution. The crowdsourcing approaches in this line of work are all highly domain-specific: we provide a general framework for using the crowd with any data processing task, and focus on improving crowd worker output quality with domain-independent techniques for macrotasks.

The crowdsourcing literature includes several frameworks and systems aimed at making it easier to build crowd-powered workflows. TurKit [20] is a library that abstracts away and caches microtask-based decisions in imperative programming flows. Frameworks like CrowdForge [16] and Dog [1] provide higher-level programming abstractions for describing crowd-powered workflows. Finally, systems like Legion [19] make it easier for developers to build applications that integrate crowd worker feedback with low latency.

As the crowdsourced work becomes more complex, researchers have proposed more sophisticated task workflows and worker organization schemes. The literature thus far on complex work has focused more on building novel workflows than on evaluating their quality. This research proposes multi-stage workflows to break work into manageable subtasks [18], maintain global constraints across multiple tasks [33], iteratively refine previous workers' efforts [20], or assemble teams of expert workers to collaborate on difficult or creative tasks [26]. Worker hierarchies have been used to organize crowdsourced managers and employees in the context of microtasks [23] and more complex work [17]. We leverage predictive modeling to increase the cost-efficiency of such hierarchies, exploring the tradeoff between increasing hierarchy depth and staying within budget constraints.

Crowds have been used to solve a variety of problems that can be addressed by macrotasks, such as structured data extraction. There are several techniques for extracting relational data from unstructured web pages [2, 4, 8] and learning automated transformations to structure and clean data [28, 14]. Additionally, crowdsourced data collection [25] and enumeration [30] can source data from the crowd.

Finally, crowdsourcing has been used in domain-independent database systems to provide structured data in response to declarative queries [11, 21, 24]. We take a similar approach to declarative crowd task and interface specification, but our

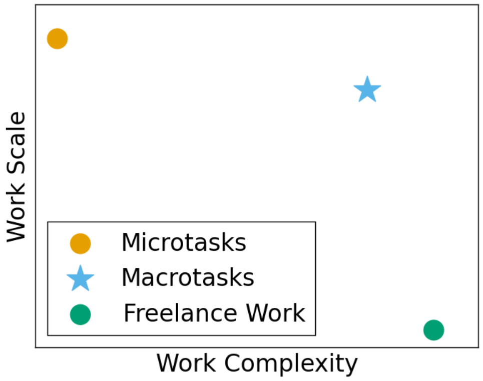


Figure 1: Tradeoffs in human-powered task completion models.

focus is on complex data processing tasks, not processing or fetching data in response to queries.

3. COMPLEX TASKS AT SCALE

Worker organization models for task completion have significant implications for the complexity and scale of the work that can be accomplished with those models. To demonstrate the need for macrotasks, we compare three organizational models: microtask-based decomposition, macrotasks, and traditional freelancer-based knowledge work. We then provide several examples of problems we solve at scale with macrotasks.

3.1 Tradeoffs with Humans in the Loop

Systems that coordinate human workers to process data make an important trade-off between complexity and scale. As work becomes increasingly complex, it requires more training and coordination of workers. As the amount of work (and therefore the number of workers) scales, the overheads associated with that coordination increase.

Figure 1 compares three forms of worker organization by their ability to handle scale and complexity. Microtasks, such as image labeling tasks sent to Amazon Mechanical Turk, are easy to scale and automate, but require effort to decompose the original high-level task into smaller microtask specifications, and are thus limited in the complexity of work they support. Traditional freelancer-based knowledge work supports arbitrarily complex tasks, because employers can interact with workers in person to convey intricate requirements and evaluate worker output. This type of work usually involves an employer personally hiring individual contractors to do a fairly large task, such as designing a website or creating a marketing campaign. The work is constrained by hiring throughput and is not amenable to automated quality control techniques, limiting its ability to scale.

Macrotasks, a middle ground between microtasks and freelance work, allow complex work to be processed at scale. Unlike microtasks, macrotasks don't require complex work to be broken down into simpler subtasks: one can assign work to workers essentially as-is, and focus on providing them with user interfaces that make them more effective. Unlike traditional knowledge work, macrotasks retain enough common structure to be specified automatically, processed uniformly in parallel, and improved in quality using automated evaluation of tasks and workers.

Much of the complex, large-scale data processing that incorporates human input is amenable to macrotask processing. Here are three high-level data-heavy use-cases we have addressed with crowd-powered macrotask workflows at a scale of millions of tasks:

1. **Structured Price List Extraction.** From Yoga studio service lists to restaurant menus, we extract structured data from PDFs, HTML, Word documents, Flash animations, and images on millions of small business websites. When possible, we automatically extract this content, but if automated extraction fails, workers must learn a complex schema and spend upwards of an hour processing the price list data for a business.

2. **Business Listings Extraction.** We extract ~30 facts about businesses (e.g., name, phone number, wheelchair accessibility) in one macrotask per business. This task could be accomplished using either microtasks or macrotasks, and we use it to help demonstrate the versatility of Argonaut.

3. **Web Design Choices.** We ask crowd workers to identify design elements such as color palettes, business logos, and other visual aspects of a website in order to enable brand-preserving transformations of website templates. These tasks are subjective and don't always have a correct answer: several color palettes might be appropriate for an organization's branding. This makes it especially challenging to judge the quality of a processed task.

The tasks above, with their complex domain-specific semantics, can be difficult to represent as microtasks, but are well-defined enough to benefit from significant automation at scale. Of course, macrotasks come with their own set of challenges, and are less thoroughly explored in the literature when compared to microtasks. There exist fewer tools for completing unstructured work, and crowd work platforms seldom offer best practices for improving the quality or efficiency of complex work. Tasks can be highly heterogeneous in their structure and output format, which makes the combination of multiple worker responses difficult and automated voting schemes for quality control nearly impossible. Macrotasks also complicate the design of worker pay structures, because payments must vary with task complexity.

3.2 A Case Study in Task Complexity

The previous discussion gave a flavor of the work we accomplish using macrotask crowdsourcing. We now describe our structured price list extraction use case in depth to demonstrate how macrotasks flow between crowd workers, and how the crowd fits in with automated data processing components. We will use this structured data extraction task as a running example throughout the paper. For simplicity, this example will focus on extraction of restaurant menus, but the same workflow applies for all price lists.

Figure 2 shows the data extraction process. Argonaut crawls small business websites or accepts price list uploads from business owners as source content from which to

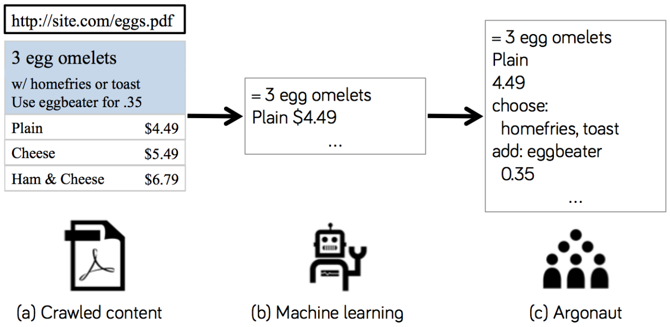
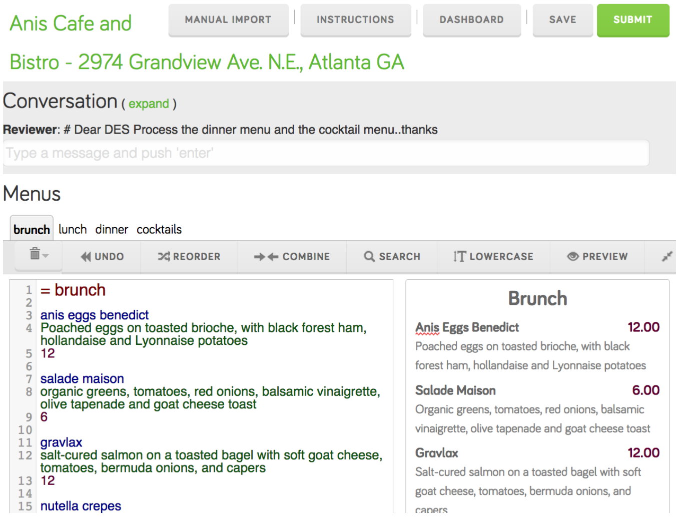


Figure 2: A crowd- and machine learning-powered workflow for extracting structured price list data.

extract price lists. Price lists come in a variety of formats, including PDFs, images, flash animations, and HTML. Automated extractors (e.g., optical character recognition, flash decompilation), and machine learned classifiers identify potential menu sections, menu item names, prices, descriptions, and item choices and additions. The output of these classifications is displayed to crowd workers in a text-based wiki markup-like format that allows fast editing of menu structure and content. Entry level crowd workers in our system, which we refer to as Data Entry Specialists (DES), correct the output of the extractors, and their work is reviewed up to two times. If automated extraction works perfectly, the crowd worker's task is simple: mark the task as being in good condition. If automated extraction fails, a crowd worker might spend up to hours manually typing all of the contents of a hard-to-extract menu. The resulting crowd-structured data is used to periodically retrain classifiers to improve their accuracy.

Our macrotask model provides for lower latency and more flexibility in throughput when compared to a freelancer model. One requirement for our use of these price list extraction tasks is the ability to handle bursts and lulls in demand. Additionally, for some tasks we require very short processing times. These constraints make a freelancer model, with slower on-boarding practices, less well suited to our problem than macrotasks.

Microtasks are also a bad fit for this price list extraction task. The tasks are complex, as workers must learn the markup format and hierarchical data schema to complete tasks, often taking 1-2 weeks to reach proficiency. Using a microtask model to complete the work would require decomposing it into pieces at a finer granularity than an individual menu. Unfortunately, the task is not easily decomposed into microtasks because of the hierarchical data schema: for example, menus contain sections which contain subsections and/or items, and prices are frequently specified not only for items, but for entire subsections or sections. There would be a high worker coordination cost if such nested information were divided across several microtasks. In addition, because raw menu text appears in a number of unstructured formats, deciding how to segment the text into items or sections for microtask decomposition would be a challenging problem in its own right, requiring machine learning or additional crowdsourcing steps. Even if microtask decomposition were successful, traditional voting-based quality control schemes would present challenges, as the free-form text in the output format can vary (e.g. punctuation, capitalization, missing/additional articles) and the schema requirements are loose. Most importantly, while it might be possible in some situations to generate hundreds of microtasks for each of the hundreds of menu items in a menu, empirical estimates based on our business process data suggest that the fair cost of a single worker on the complex version of these tasks is significantly lower than the redundant version of the many microtasks it would take to process most menus.



Anis Cafe and Bistro - 2974 Grandview Ave. N.E., Atlanta GA

Conversation (expand)

Reviewer: # Dear DES Process the dinner menu and the cocktail menu..thanks

Type a message and push 'enter'

Menus

```
= brunch

anis eggs benedict
Poached eggs on toasted brioche, with black forest ham,
hollandaise and Lyonnaise potatoes
12

salade maison
organic greens, tomatoes, red onions, balsamic vinaigrette,
olive tapenade and goat cheese toast
6

gravlax
salt-cured salmon on a toasted bagel with soft goat cheese,
tomatoes, bermuda onions, and capers
12

nutella crepes
```

Brunch

Anis Eggs Benedict 12.00
Poached eggs on toasted brioche, with black forest ham, hollandaise and Lyonnaise potatoes

Salade Maison 6.00
Organic greens, tomatoes, red onions, balsamic vinaigrette, olive tapenade and goat cheese toast

Gravlax 12.00
Salt-cured salmon on a toasted bagel with soft goat cheese, tomatoes, bermuda onions, and capers

Figure 3: The Argonaut framework crowd worker user interface on a price list extraction task. The interface font is artificially enlarged for readability.

We have presented several examples of crowd work suitable for macrotask processing, and taken an in-depth look at one structured data extraction task that is not amenable to either microtask or freelance knowledge work-style processing. In the following sections, we describe the system we have designed for implementing the price lists task and other macrotask workflows, focusing specifically on the challenges of improving work quality in complex tasks.

4. THE ARGONAUT FRAMEWORK

This section provides an overview of Argonaut, a framework that combines automated models with complex crowd tasks. Argonaut is a scheme for quality control in macrotasks that can generalize across many applications in the presence of heterogeneities task outputs. We have used Argonaut for performing several data processing tasks, but will use structured data extraction as our running example. To reduce error introduced by crowd workers while remaining domain-independent, the framework uses three complementary techniques that we describe next: a review hierarchy, predictive task modeling, and worker modeling. These techniques are effective when dealing with tasks that are complex and highly context-sensitive, but still have structured output.

Figure 3 shows Argonaut as experienced by a crowd worker on a price list extraction task. The *Menu* section is designed by the Argonaut user/developer. The rest of the interface is uniform across all task types, including a *Conversation* box for discussion between crowd workers.

4.1 Framework API

To define a new macrotask type, a developer using Argonaut provides the following information:

Task data. Users must implement a method that provides task-specific data encoded as JSON for each task. Such data might be serialized in various ways. For example, business listings tasks produce a key-value mapping of business attributes (e.g., phone numbers, addresses). For price lists, a markup language allows workers to edit blocks of text and label them (e.g., sections, menu items).

Worker interface renderer. Given task data, users must implement a method that generates an HTML `<div>` element with a worker user interface. Here is an example rendering of menu data:

```
def get_render_html():
  return """
  <div>
    <p>Edit the text according to the
    <a href="guidelines.html">guidelines.</a>
    Please structure
    <a href="{{menu_url}}">this menu.</a></p>
    <form><textarea name="structured_menu"
           value="{{data.menu_text}}"></form>
  </div>"""
```

Other interface features (e.g., a commenting interface for workers to converse, buttons to accept/reject a task) are common across different task types and provided by Argonaut.

Error metric. Given two versions of task data (e.g., an initial and a reviewed version), an error metric helps the TaskGrader (Section 4.4) determine how much that task has changed. For textual data, this metric might be based on the number of lines changed, whereas more complex metrics are required for media such as images or video. Users can pick from Argonaut's pre-implemented error metrics or provide one of their own.

Users adding a new macrotask type to Argonaut need not write any backend code to manage tasks or workers. They simply build the user interface for the task workflow and wire it up to Argonaut's API.

4.2 Framework Architecture

We now describe Argonaut's main components by following the path of a task through the framework as depicted in Figure 4. First, a requester submits tasks to the system. The requester specifies tasks and the workers' user interface using the framework API described in Section 4.1. Newly submitted tasks go to the Task Manager, which can send tasks to the crowd for processing. The Task Manager receives tasks that have been completed by crowd workers, and decides if those tasks should go back to the crowd for subsequent review, or be returned to the requester as a finalized task. The Task Manager uses the TaskGrader model, which predicts the amount of error remaining in a task (Section 4.4), to make this decision. If the model predicts that a high amount of error remains in the task, the task will require an additional review from the crowd. When a task is sent to the crowd, the Task Manager specifies which expertise level in the review hierarchy should process the task. Tasks that are newly submitted by a requester are assigned to the lowest level in the hierarchy, to be processed by workers known as Data Entry Specialists. From the Task Manager, tasks go to the Worker Manager. The Worker Manager manages the crowd workers and determines which worker within the assigned hierarchy level to route a task to. Figure 5 shows a more detailed view of the hierarchy (Section 4.3).

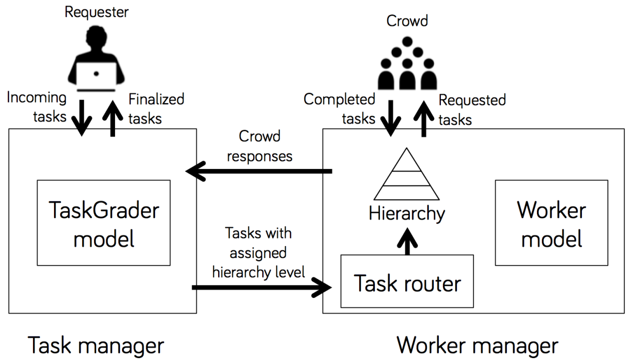


Figure 4: The Argonaut framework architecture for macrotask data processing.

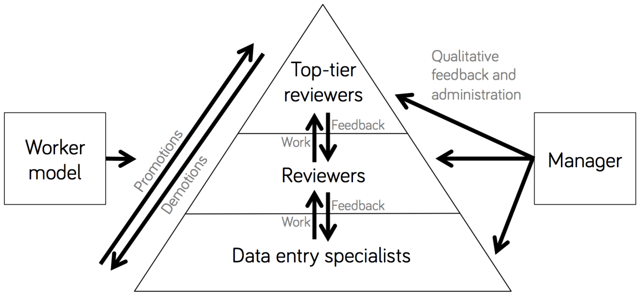


Figure 5: The hierarchy of task review. Trusted workers review entry-level workers' output and provide low-level feedback on tasks, managers provide high-level feedback to every worker, and a model of worker speed and accuracy chooses workers to promote and demote throughout the hierarchy.

4.3 Review Hierarchy

To achieve high task quality, Argonaut identifies a crowd of trusted workers and organizes them in a hierarchy with the most trusted workers at the top. Workers that perform well review the output of less trusted workers. High quality is achieved through review, corrections, and recommendations of educational content to entry-level workers.

The review hierarchy is depicted in Figure 5. Workers at the bottom level are referred to as Data Entry Specialists (DES). DES workers generally have less experience, training, and speed than the Reviewer-level workers. They are the first to see a task and do the bulk of the work. In the case of structured data extraction, a DES sees the output of automated extractors, and might either approve of a high-quality extraction or spend up to a few hours manually inputting or correcting the results of a failed automated extraction. Reviewers review the work of the DES, and the best Reviewers review the work of other Reviewers. As a worker's output quality improves, less of their work is reviewed.

Developing a trusted crowd requires significant investment in on-boarding and training. In our experience, on-boarding a DES requires that they spend several days studying a text- and example-heavy guide on our price list syntax. The worker must pass a qualification quiz before she or he can complete tasks. A newly hired worker has a trial period of 4 weeks, during which every task they complete is reviewed. Because the training examples can not cover all real-life possibilities, feedback and additional on-the-job training from more experienced workers is essential to developing the DES. Reviewers examine the DES's work and provide detailed feedback in the form of comments and edits. They can reject the task and send it back to the DES, who must make corrections and resubmit. This workflow allows more experienced workers to pass on their knowledge and experience. By the end of the trial period, enough data has been collected to evaluate the worker's work quality and speed.

Because per-task feedback only provides one facet of worker training and development, Argonaut relies on a crowd Manager to develop workers more qualitatively. This Manager is manually selected from the highest quality Reviewers, and handles administrative tasks while fielding questions from other crowd workers. The Manager also looks for systemic misunderstandings that a worker has, and sends personalized emails suggesting improvements and further reading. Workers receive such a feedback email at least once per month. In reviewing workers, the Manager also recommends workers for promotion/demotion, and this feedback contributes to hierarchy changes. If the Manager spots an issue that is common to several workers, the Manager might generate a new training document to supplement workers' education. Although the crowd hierarchy is in this way self-managing, the process of on-boarding users and ending contracts is not left to the Manager: it requires manual intervention by the framework user.

Workers are incentivized to complete work quickly and at a high level of quality. A worker's speed and quality rankings are described in more detail in Section 4.5, but in short, workers are ranked by how poorly they performed in their middling-to-worst tasks, and by how quickly they completed tasks relative to other workers. Given this ranking, workers are promoted or demoted appropriately on a regular basis. Reviewers are paid an hourly wage, while DES are paid a fixed rate based on the difficulty of their task, which can be determined after a reviewer ensures that they have done their work correctly. This payment mechanism incentivizes Reviewers to take the time they need to give workers meaningful feedback, while DES are incentivized to complete their tasks at high quality as quickly as possible. Based on typical work speed of a DES, Reviewers receive a higher hourly wage. The Manager role is also paid hourly, and earns the highest amount of all of the crowd workers. As a further incentive to do good work quickly, workers are rate-limited per week based on their quality and speed over the past 28 days. For example, the top 10% of workers are allowed to work 45 hours per week, the next 25% are allowed 35 hours, and so on, with the worst workers limited to 10 hours.

4.4 TaskGrader

A predictive model, called TaskGrader, decides which tasks to review. TaskGrader leverages available worker context, work history, and past reviews to train a regression model that predicts an error score used to decide which tasks are reviewed. The goal of the TaskGrader is to maximize quality, which we measure as the number of errors caught in a review. The space of possible implementations of TaskGrader spans three objectives:

- *Throughput* is the total number of tasks processed. For the design of TaskGrader, we hold throughput constant and view the initial processing of each task as a fixed cost.
- *Cost* is the amount of human effort spent by the system measured in tasks counts. We hold this constant at an average of 1.56 workers per task (a parameter which should be set based on available budget and throughput requirements). The TaskGrader can allocate either 1, 2, or 3 workers per task, subject to the constraint that the average is 1.56.
- *Quality* is the inverse of the number of errors per task. Quality is difficult to measure in absolute terms, but can be viewed as the steady state one would reach by applying infinite number of workers per task. We approximate quality by the number of changes (which we assume to be errors fixed) made by each reviewer. The goal of the TaskGrader is to maximize the amount of errors fixed across all reviewed tasks.

In order to generate ground truth training data for our supervised regression model, we take advantage of past data from the hierarchical review model. We use the fraction of output lines of a task that are incorrect as an error metric. We approximate this value by measuring the lines changed by a subsequent reviewer of a task. We compute training labels by measuring the difference between the output of a task before and after review. Thus, tasks that have been reviewed in the hierarchy are usable as labeled examples for training the model.

The TaskGrader uses a variety of data collected on workers as features for model training. Table 1 describes and categorizes the features we use. We find it useful to categorize features into two groupings:

- How *task-specific* (e.g., how long did a task take to complete) or how *worker-specific* (e.g., how has the worker done on the past few tasks) is a feature? A common approach to ensuring work quality in microtask frameworks is to identify the best workers and provide them with the most work. We use this categorization to measure how predictive of work quality the worker-specific features were.
- Is a feature *generalizable* across task types (e.g., the time of day a worker is working) or is it *domain-specific*

Feature Name or Group	Description	Categorization	
percent of input changed	how much of the task a worker changed from the input they saw	task-specific	domain-specific
grammar and spelling errors	errors such as misspellings, capitalization mistakes, and missing commas	task-specific	domain-specific
domain-specific automatic validation	errors detected by automatic checkers such as very high prices, duplicate price lists, missing prices	task-specific	domain-specific
price list statistics	statistics on task output like # of price lists, # of sections, # items per section, price list length	task-specific	domain-specific
task times of day	time of day when different stages of the workflow are completed	task-specific	generalizable
processing time	time it took for a worker to complete the task	task-specific	generalizable
task urgency	high priority tasks must be completed within a certain time and can not be rejected	task-specific	generalizable
tasks per week	# of tasks completed per week over past few weeks	worker-specific	generalizable
distribution of past task error scores	deciles, mean, std dev, kurtosis of past error scores	worker-specific	generalizable
distribution of speed on past tasks	deciles, mean, std dev, kurtosis of past processing times	worker-specific	generalizable
worker timezone	timezone where worker works	worker-specific	generalizable

Table 1: Descriptions of TaskGrader Features. Each row represents one or more features. The *Categorization* column places features into broad groups that will be used to evaluate feature importance.

(e.g., processing a pizza menu vs. a sushi menu)? We are interested specifically in how predictive the generalizable feature set is, because generalizable features are those that could be used in any crowd system, and would thus be of larger interest to an organization wishing to employ a TaskGrader-like model.

We use an online algorithm for selecting tasks to review, because new tasks continuously arrive on our system. This online algorithm frames the problem as a regression: the TaskGrader predicts the amount of error in a task, having dynamically set a review threshold at runtime in order to review tasks with the highest error without overrunning the available budget. If we assumed a static pool of tasks, the problem might better be expressed as a ranking task [7].

To ensure a consistent review budget (e.g., 40% of tasks should be reviewed), we must pick a threshold for the TaskGrader regression so that we spend the desired budget on review. Depending on periodic differences in worker performance and task difficulty, this threshold can change. Every few hours, we load the TaskGrader score distribution for the past several thousand tasks and empirically set the TaskGrader review threshold to ensure that the threshold would have identified the desired number of tasks for review. In practice, this procedure results in accurate TaskGrader-initiated task review rates.

We must be careful with the tasks we pick for future TaskGrader training. Because tasks selected for review by the TaskGrader are biased toward high error scores, we can not use them to unbiasedly train future TaskGrader models. We reserve a fraction of our overall review budget to randomly select tasks for review, and train future TaskGrader models on only this data. For example, if we review 30% of tasks, we aim to have the TaskGrader select the worst 25% of tasks, and select another 5% of tasks for review randomly, only using that last 5% of tasks to train future models.

Occasionally users of the system may need to apply domain-specific tweaks to the error score. We initially presented the task error score as the fraction of the output lines that was found incorrect in review. In its pure form, the score should lend itself reasonably well to various text-based complex work. However, one must be careful that the error score is truly representative of high or low quality. In our scenario, workers can apply comments throughout a price list's text to explain themselves without modifying the displayed price list content (e.g., "# I couldn't find a menu on this website, leaving task empty"). Reviewers sometimes changed the comments for readability, causing the comments to appear as line differences, thus affecting the error score. These comments are not relevant to the output, so we were penalizing workers for differences that we did not care about. For near-empty price lists, this had an especially strong effect on the error score and skewed our results. When we modified the system to remove comments prior to computing the error score, we found anecdotally that the accuracy rose by nearly 5%.

4.5 Modeling the Crowd

A key aspect of Argonaut is the ability to identify skilled workers to promote to reviewer status. In order to identify which crowd workers to promote near the top of the hierarchy, we have developed a metric by which we rank all workers, composed of two components:

- *Work quality.* Given all of the tasks a worker has completed recently, we take the error score of their 75^{th} percentile worst score. In Section 5, we show that worker error percentiles around 80% are the most important *worker-specific* feature for determining the quality of a task. We sort all workers by their 75^{th} percentile error score, and assign each worker a score from 0 (worst) to 1 (best) based on this ranking.

- *Work speed.* We measure how long each worker takes to complete tasks on average. We rank all workers by how quickly they complete tasks, assigning workers a score from 0 (worst) to 1 (best) based on this ranking.

We then take a weighted average of these two metrics as our worker quality measure. With this overall score for each worker, we can promote, demote, provide bonuses to, or end contracts with workers depending on overall task availability. In practice, because we rate-limit the lowest-performing workers if we are going to overrun our weekly budget, we rarely have to explicitly end a contract.

Our system does not currently utilize statistical spam detection techniques. However, promotion/demotion incentives, Manager feedback, and long-term relationships with crowd workers are effective at keeping malicious workers out of the hierarchy.

4.6 Crowd Output as Training Data

We describe a structured data extraction workflow in Section 3.2. Since macrotasks power its crowd component, and because the automated extraction and classifiers do not hit good enough precision/recall levels to blindly trust the output, at least one crowd worker looks at the output of each automated extraction. In this scenario, there is still benefit to a crowd-machine hybrid: because crowd output takes the same form as the output of the automated extraction, our extraction techniques can learn from crowd relabeling. As they improve, the system requires less crowd work for high-quality results. This active learning loop applies to any data processing task with iteratively improvable output: one can train a learning algorithm on the output of a reviewed task, and use the model to classify future tasks before humans process them in order to reduce manual worker effort.

4.7 Crowd on-boarding and Maintenance

Though Argonaut handles the systems challenges of processing and reviewing macrotasks, building a crowd hierarchy based on workers recruited from platforms like oDesk[1] still requires manual intervention. Bootstrapping the hierarchy takes significant investment. To start developing our system, we initially brought on about 20 contractors from oDesk as DES workers, and an in-house employee reviewed their work. Over time, we promoted fast and accurate crowd workers to Reviewer status, and our full time employee only reviewed the Reviewers. Eventually, we selected our top Reviewer to be a Manager, and now full-time employees only communicate with the crowd when a bug is discovered.

Once the initial hierarchy has been trained and assembled, however, growing the hierarchy or adapting it to new macrotask types is efficient. Managers streamline the development of training materials, and although new workers require time to absorb documentation and work through examples, this training time is significantly lower than the costs associated with the traditional freelance knowledge worker hiring process.

5. EXPERIMENTS

In this section, we evaluate the impact of the techniques proposed in Section 4 on reducing error in macrotasks and investigate whether these techniques can generalize to other applications. We base our evaluations on a crowd workflow that has run in industry for almost 3 years, has processed over 1.3 million tasks, and has handled over half a million hours of human contributions, primarily for the purpose of doing large-scale structured web data extraction. We show that reviewers improve most tasks they touch, and that workers higher in the hierarchy spend less time on each task. We find that the TaskGrader focuses reviews on tasks with considerably more errors than random spot-checking. We then train the TaskGrader on varying subsets of its features and show that domain-independent (and thus generalizable) features are sufficient to significantly improve the workflow's data quality, supporting the hypothesis that such a model can add value to any macrotask crowd workflow with basic logging of worker activity. We additionally show that at constrained review budgets, combining the TaskGrader and a multilayer review hierarchy uncovers more errors than simply reviewing more tasks in single-level review. Finally, we show that a second phase of review often catches errors in a different set of tasks than the first phase.

5.1 System Operation

In a little under three years, we have developed a trained crowd of ~300 workers, which has spiked to almost 1000 workers at various times to handle increased throughput demands. Currently, the crowd's composition is approximately 78% DES, 12% Reviewers, and 10% top-tier Reviewers. Top-tier Reviewers can review anyone's output, but typically review the work of other Reviewers to ensure full accountability. More than 50% of workers have contributed to our system for at least 2.5 years, and 80% of workers have contributed for more than 1.5 years.

The Manager sends 5-10 emails a day to workers with specific issues in their work, such as spelling/syntax errors or incorrect content. He also responds to 10-20 emails a day from workers with various questions and comments.

The throughput of the system varies drastically in response to business objectives. The 90th percentile week saw 19k tasks completed, and the 99th percentile week saw 33k tasks completed, not all of which were structured data extraction tasks. Tasks are generally completed within a few hours, and 75% of all tasks are completed within 24 hours.

5.2 Dataset

We evaluate our techniques on an industry deployment of Argonaut, in the context of the complex price list structuring task described in Section 3.2. The crowd forming the hierarchy is described in Section 4.3. The training data consisted of a subset of approximately 60k price list-structuring tasks that had been spot-checked by Reviewers over a period of one year. Most tasks corresponded to a business, and the worker is expected to extract all of the price lists for that business. The task error score distribution is heavily skewed toward 0: 62% of tasks have an error score less than 0.025. If the TaskGrader could predict these scores, we could decrease review budgets without affecting output quality. 27% of the tasks contain no price lists and result in empty output. This happens if, for example, the task links to a website that does not exist, or doesn't contain any price lists. For these tasks, the error score is usually either 0 or 1, meaning the worker correctly identified that the task is empty, or they did not.

5.3 Time Spent in the Hierarchy

Figure 6 shows the amount of time workers spend at various stages of task completion. The initial phase of work might require significant data entry if automated extraction fails, and varies depending on the length of the website being extracted. This phase generally takes less than an hour, but

[1] http://www.odesk.com

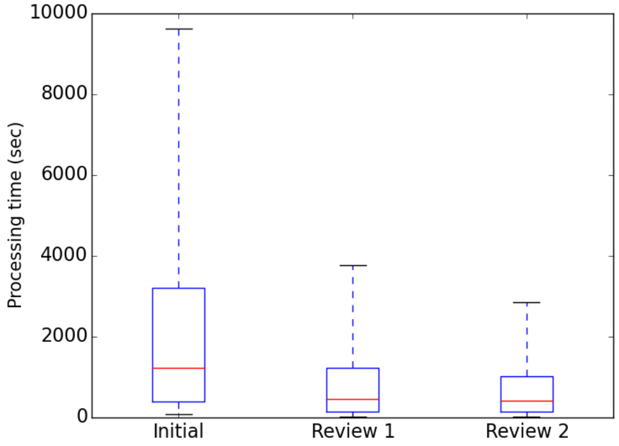


Figure 6: The distribution of processing times for price list tasks, broken down by the initial task, the first review, and the second review. Times are at 30-second granularity. Red line represents the median. Box represents the 25 to 75th percentiles. Whiskers represent 5 and 95th percentiles.

Metric Name	Count	Percentage
Total tasks	75	-
Discarded tasks	4	-
Valid tasks	71	100%
Decreased quality	7	9.9%
No discernible change	13	18.3%
Improved quality	51	71.8%

Table 2: Of the 71 valid tasks two authors coded, 9.9% decreased in quality after review, 18.3% had no discernible change, and 71.8% improved in quality.

can take up to three hours in the worst case. Subsequent review phases take less time, with both phases generally taking less than an hour each. Review 1 tasks generally take longer than Review 2 tasks, likely because: 1) we promote workers that produce high quality work quickly, and so Review 2 workers tend to be faster, and 2) if Review 1 catches errors, Review 2 might require less work.

5.4 Effectiveness of Review

We evaluate the effectiveness of review in several ways, starting with expert coding. Two authors looked at a random sample of 50 tasks each that had changed by more than 5% in their first review. The authors were presented with the pre-review and post-review output in a randomized order so that they could not tell which was which. For each task, the authors identified which version of the task, if any, was of higher quality. The two sets of 50 tasks overlapped by 25 each, so that we could measure agreement rates between authors, and resulted in 75 unique tasks for evaluation.

For the 25 tasks on which authors overlapped, two were discarded because the website was no longer accessible. Of the remaining 23 tasks, authors agreed on 21 of them, with one author marking the remaining 2 as indistinguishable in quality. Given that authors agreed on all of the tasks on which they were certain, we find that expert task quality coding can be a high-agreement activity.

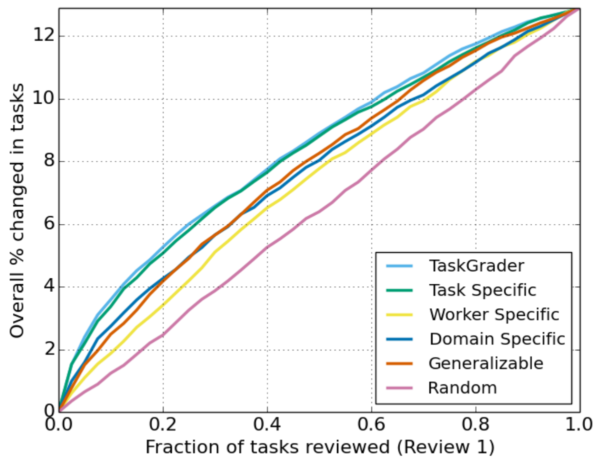


Figure 7: Cumulative percentage of each task changed divided by total number of tasks for TaskGrader models trained on various subsets of features, with random review provided as a baseline. This figure contains Review 1 findings only, with Review 2 performance excluded. Descriptions of which features fall into the Task Specific, Worker Specific, Domain Specific, and Generalizable categories can be found in Table 1.

Table 2 summarizes the results of this expert coding experiment. Of 75 tasks, 4 were discarded for technical reasons (e.g., website down). Of the remaining 71, the authors found 13 to not be discernibly different in either version. On 51 of the tasks, the authors agreed that the reviewed version was higher-quality (though they were blind to which task had been reviewed when making their choice). This suggests that, on our data thresholded by $\geq 5\%$ of lines changed, we found that review decreases quality 9.9% of the time, does not discernibly change quality 18.3% of the time, and improves quality 71.8% of the time.

These findings point toward the key benefit of the hierarchy: when a single review phase causes a measurable change in a task, it improves output with high probability.

5.5 TaskGrader Performance

Since task quality varies, it is important for the TaskGrader to identify the lowest-quality tasks for review. We trained the TaskGrader, a gradient boosting regression model, on 90% of the data as a training set, holding out 10% as a test set. We compared gradient boosting regression to several models, including support vector machines, linear regression, and random forests, and used cross-validation on the training set to identify the best model type. We also used the training set to perform a grid search to set hyperparameters for our models.

We evaluate the TaskGrader by the aggregate errors it helps us catch at different review budgets. To capture this notion, we compute the errors caught (represented by the

Review Budget	20%	40%	60%	80%	100%
Optimal % reviewed twice	14.3	14.3	14.3	14.3	29.0
% improvement over random	118	53.6	35.3	21.4	16.2

Table 3: Improvement over random spot-checks with optimal Review 1 and Review 2 splits at different budgets.

percentage of lines changed in review) by reviewing the tasks identified by the TaskGrader. We compare these to the errors caught by reviewing a random sample of N percent of tasks. Figure 7 shows the errors caught as a function of fraction of tasks reviewed for the TaskGrader model trained on various feature subsets, as well as a baseline random review strategy. We find that at all review budgets less than the trivial 100% case (wherein the TaskGrader is identical to random review), the TaskGrader is able to identify significantly more error than the random spot check strategy.

5.6 TaskGrader Generalizability

We now simultaneously explore which features are most predictive of task error and whether the model might generalize to other problem areas. In Section 4.4, we broke the features used to train the TaskGrader into two groupings: task-specific vs worker-specific, and generalizable vs. domain-specific. We now study how these groupings affect model performance.

Figure 7 shows the performance of the TaskGrader model trained only on features from particular feature groupings. Each feature grouping performs better than random sampling, suggesting they provide some signal.

Generalizable features perform comparably to domain-specific ones. Because features unrelated to structured data extraction are still predictive of task error, it is likely that the TaskGrader model can be implemented easily in other macrotask scenarios without losing significant predictive power.

For our application, it is also interesting to note that task-specific features, such as work time and percent of input changed, outperform worker-specific features, such as mean error on past tasks. This finding is counter to the conventional wisdom on microtasks, where the primary approaches to quality control rely on identifying and compensating for poorly-performing workers. There could be several reasons for this difference: 1) over time, our incentive systems have biased poorly performing workers away from the platform, dampening the signal of of individual worker performance, and 2) there is high variability in macrotask difficulty, so worker-specific features do not capture these effects as well as task-specific ones.

5.7 TaskGrader in a Hierarchy

The TaskGrader is applied at each level of the hierarchy to determine if the task should be sent to the next level. Figure 8 shows the error caught by using the TaskGrader to send tasks for a first and second review. The maximum percent changed (at 1.0 on the x-axis) is smaller in Review 2 than in Review 1, which suggests that tasks are higher quality on average by their second review, therefore requiring fewer improvements.

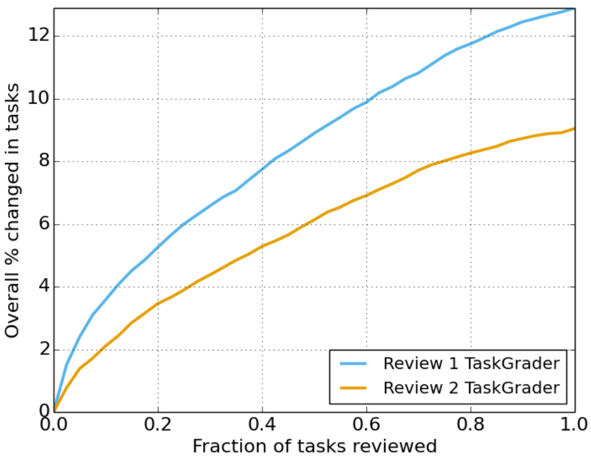


Figure 8: Cumulative percentage of each task changed divided by total number of tasks for TaskGrader in both phase one and phase two of review.

We also examined how the amount of error caught would change if we split our budget between Review 1 and Review 2, using the TaskGrader to help us judge if we should review a new task (Review 1), or review a previously reviewed task (Review 2). This approach might catch more errors by reviewing the worst tasks multiple times and not reviewing the best tasks at all. Figure 9 shows the total error caught for a fixed total budget as we vary the split between Review 1 and Review 2. The budget values shown in the legend are the number of tasks that get reviews as a percentage of the total number of tasks in the system. The x-axis ranges from 0% Review 2 (100% Review 1) to 100% Review 2. Since a task can not see Review 2 without first seeing Review 1, 100% Review 2 means the budget is split evenly between Review 1 and Review 2. For example, if the budget is an average of 0.4 reviews per task, at the 100% Review 2 data point, 20% of tasks are selected for both Review 1 and Review 2.

Examining the figure, we see that for a given budget, there is an optimal trade-off between level 1 and level 2 review. Table 3 shows the optimal percent of tasks to review twice along with the improvement over random review at each budget. As the review budget decreases, the benefit of TaskGrader-suggested reviews become more pronounced, yielding a full 118% improvement over random at a 20% budget. It is also worth noting that with a random selection strategy, there is no benefit to second-level review: on average, randomly selecting tasks for a second review will catch fewer errors than simply reviewing a new task for the first time (as suggested by Figure 8).

5.8 Effectiveness of Subsequent Reviews

Next we examine in more detail what is being changed by the two phases of review. We measure if reviewers are editing the same tasks and also how correlated the magnitude of the Review 1 and Review 2 changes are.

In order to measure the overlap between the most changed tasks in the two phases of review, we start with a set of 39,180 tasks that were reviewed twice. If we look at the

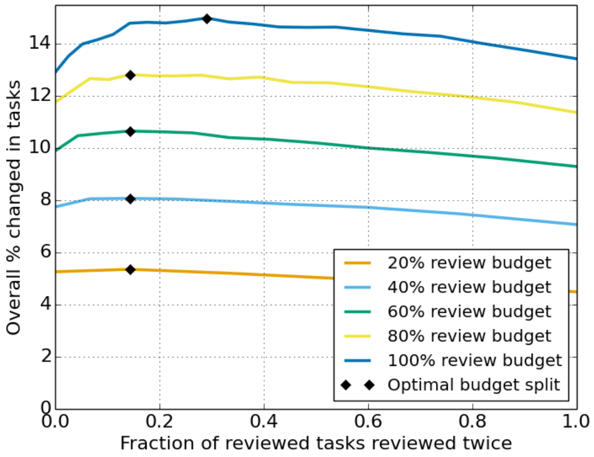


Figure 9: Cumulative percentage of each task changed divided by total number of tasks for different budgets of total reviews. The left side represents spending 100% of the budget on phase one, the right side represents splitting the budget 50/50 and reviewing half as many tasks two times each.

20% (approx. 7840) most changed tasks in Review 1 and the 20% most changed tasks in Review 2, the two sets of tasks overlap by around 25% (approx. 1960). We leave out the full results due to space restrictions, but this trend continues in that the most changed tasks in each phase of review do not meaningfully overlap until we look at the 75% most changed tasks in each phase. This suggests that Review 2 errors are mostly caught in tasks that were not heavily corrected in Review 1.

As another measure of the relationship between Review 1 and Review 2, we measure the correlation between the percentage of changes to a task in each review phase. The Pearson's correlation, which ranges from -1 (completely inverted correlation) to 1 (completely positive correlation), with 0 representing no correlation, was 0.096. To avoid making distribution assumptions about our data, we also measured the nonparametric Spearman's rank correlation and found it to be 0.176. Both effects were significant with a two-tailed p-value of $p < .0001$. In both cases, we find a very weak positive correlation between the two phases of review, which suggests that while Review 1 and Review 2 might correct some of the same errors, they largely catch errors on different tasks.

These findings support the hierarchical review model in an unintuitive way. Because we know review generally improves tasks, it is interesting to see two serial review phases catching errors on different tasks. This suggests some natural and exciting follow-on work. First, because Review 2 reviewers are generally higher-ranked, are they simply more adept at catching more challenging errors? Second, are the classes of errors that are caught in the two phases of review fundamentally different in some way? Finally, can the overlap be explained by a phenomenon such as "falling asleep at the wheel," where reviewer attention decreases over the course of a sitting, and subsequent review phases simply provide more eyes and attention? Studying deeper review hierarchies and classifying error types will be interesting future work to help answer these questions.

6. CONCLUSIONS

Our results show that in crowd workflows built around macrotasks, a worker hierarchy, predictive modeling to allocate reviewing resources, and a model of worker performance can effectively reduce error in task output. As the budget available to spend on task review decreases, these techniques are both more important and more effective, combining to provide up to 118% improvement in errors caught over random spotchecking. While our features included a mix of domain-specific and generalizable features, using only the generalizable features resulted in a model that still had significant predictive power, suggesting that the Argonaut hierarchy and TaskGrader model can easily be trained in other macrotask settings without much task-specific featurization. The approaches that we present in this paper are used at scale in industry, where our production implementation significantly improves data quality in a crowd work system that has handled millions of tasks and utilized over half a million hours of worker participation.

7. ACKNOWLEDGEMENTS

We are grateful to the nearly 1000 crowd workers that made our work possible, and to several people who made Argonaut what it is: Zain Allarakhia, Solomon Bisker, Silas Boyd-Wickizer, Peter Downs, Matthew Greenstein, Kainar Kamalov, Emma Lubin, Arsen Mamikonyan, Usman Masood, Keir Mierle, Marek Olszewski, Marc Piette, Rene Reinsberg, Stelios Sidiroglou-Douskos, and Maksim Stepanenko. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE 1106400.

8. REFERENCES

[1] S. Ahmad et al. The jabberwocky programming environment for structured social computing. In *UIST*, 2011.

[2] A. Arasu, H. Garcia-Molina, and S. University. Extracting structured data from Web pages. In *the 2003 ACM SIGMOD international conference on*, pages 337–348, New York, New York, USA, 2003. ACM Press.

[3] A. Arasu, M. Götz, and R. Kaushik. On active learning of record matching packages. In *SIGMOD '10: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM Request Permissions, June 2010.

[4] M. Banko, M. J. Cafarella, S. Soderland, M. Broadhead, and O. Etzioni. Open information extraction from the web. *IJCAI '07*, pages 2670–2676, Jan. 2007.

[5] K. Bellare, S. Iyengar, A. G. Parameswaran, and V. Rastogi. Active sampling for entity matching. In *KDD '12: Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM Request Permissions, Aug. 2012.

[6] M. S. Bernstein, G. Little, R. C. Miller, B. Hartmann, M. S. Ackerman, D. R. Karger, D. Crowell, and K. Panovich. Soylent: a word processor with a crowd inside. In *UIST '10: Proceedings of the 23nd annual ACM symposium on User interface software and technology*. ACM Request Permissions, Oct. 2010.

[7] C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. Hullender. Learning to rank using gradient descent. In *ICML '05: Proceedings of the 22nd international conference on Machine learning*. ACM, Aug. 2005.

[8] M. J. Cafarella, A. Halevy, D. Z. Wang, E. Wu, and Y. Zhang. WebTables. *Proceedings of the VLDB Endowment*, 1(1):538–549, Aug. 2008.

[9] A. Das Sarma, A. Jain, A. Machanavajjhala, and P. Bohannon. An automatic blocking mechanism for large-scale de-duplication tasks. In *CIKM '12: Proceedings of the 21st ACM international conference on Information and knowledge management*. ACM Request Permissions, Oct. 2012.

[10] G. Demartini, D. E. Difallah, and P. Cudré-Mauroux. ZenCrowd: leveraging probabilistic reasoning and crowdsourcing techniques for large-scale entity linking. In *WWW '12: Proceedings of the 21st international conference on World Wide Web*. ACM, Apr. 2012.

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

[12] C. Gokhale, S. Das, A. Doan, J. F. Naughton, N. Rampalli, J. Shavlik, and X. Zhu. Corleone: Hands-Off Crowdsourcing for Entity Matching. *SIGMOD 2014*, Mar. 2014.

[13] P. G. Ipeirotis, F. Provost, and J. Wang. Quality management on Amazon Mechanical Turk. In *HCOMP '10: Proceedings of the ACM SIGKDD Workshop on Human Computation*. ACM Request Permissions, July 2010.

[14] S. Kandel, A. Paepcke, J. M. Hellerstein, and J. Heer. Wrangler, Jan. 2011.

[15] D. R. Karger, S. Oh, and D. Shah. Iterative Learning for Reliable Crowdsourcing Systems. *Advances in neural information . . .*, pages 1953–1961, 2011.

[16] A. Kittur, B. Smus, S. Khamkar, and R. E. Kraut. Crowdforge: crowdsourcing complex work. In *UIST*, pages 43–52, 2011.

[17] A. Kulkarni, P. Gutheim, P. Narula, D. Rolnitzky, T. S. Parikh, and B. Hartmann. MobileWorks: Designing for Quality in a Managed Crowdsourcing Architecture. *IEEE Internet Computing ()*, 16(5):28–35, 2012.

[18] A. P. Kulkarni, M. Can, and B. Hartmann. Collaboratively crowdsourcing workflows with turkomatic. *CSCW*, pages 1003–1012, 2012.

[19] W. S. Lasecki, K. I. Murray, S. White, R. C. Miller, and J. P. Bigham. Real-time crowd control of existing interfaces. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, pages 23–32. ACM, 2011.

[20] G. Little, L. B. Chilton, M. Goldman, and R. C. Miller. TurKit: tools for iterative tasks on mechanical Turk. *KDD Workshop on Human Computation*, pages 29–30, 2009.

[21] A. Marcus, E. Wu, D. Karger, S. Madden, and R. Miller. Human-powered sorts and joins. In *Proceedings of the VLDB Endowment*. VLDB Endowment, Sept. 2011.

[22] B. Mozafari, P. Sarkar, M. J. Franklin, M. I. Jordan, and S. Madden. Active Learning for Crowd-Sourced Databases. *arXiv.org*, Sept. 2012.

[23] J. Noronha, E. Hysen, H. Zhang, and K. Z. Gajos. Platemate: crowdsourcing nutritional analysis from food photographs. In *UIST '11: Proceedings of the 24th annual ACM symposium on User interface software and technology*. ACM Request Permissions, Oct. 2011.

[24] A. G. Parameswaran, H. Park, H. Garcia-Molina, N. Polyzotis, and J. Widom. Deco: declarative crowdsourcing. In *CIKM '12: Proceedings of the 21st ACM international conference on Information and knowledge management*. ACM Request Permissions, Oct. 2012.

[25] H. Park and J. Widom. CrowdFill: collecting structured data from the crowd. *SIGMOD*, pages 577–588, 2014.

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

[27] J. M. Rzeszotarski and A. Kittur. Instrumenting the crowd: using implicit behavioral measures to predict task performance. In *UIST '11: Proceedings of the 24th annual ACM symposium on User interface software and technology*. ACM Request Permissions, Oct. 2011.

[28] S. Soderland. Learning Information Extraction Rules for Semi-Structured and Free Text. *Machine Learning*, 34(1/3):233–272, 1999.

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

[30] B. Trushkowsky, T. Kraska, M. J. Franklin, and P. Sarkar. Crowdsourced enumeration queries. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 673–684, 2013.

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

[32] S. E. Whang, P. Lofgren, and H. Garcia-Molina. Question selection for crowd entity resolution. In *Proceedings of the VLDB Endowment*. VLDB Endowment, Apr. 2013.

[33] H. Zhang, E. Law, R. Miller, K. Gajos, D. Parkes, and E. Horvitz. Human computation tasks with global constraints. In *CHI '12: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM Request Permissions, May 2012.