Turkalytics: Analytics for Human Computation

Paul Heymann
Department of Computer Science
Stanford University
Stanford, CA, 94305, USA
heymann@stanford.edu

Hector Garcia-Molina
Department of Computer Science
Stanford University
Stanford, CA, 94305, USA
hector@cs.stanford.edu

ABSTRACT

We present "Turkalytics," a novel analytics tool for human computation systems. Turkalytics processes and reports logging events from workers in real-time and has been shown to scale to over one hundred thousand logging events per day. We present a state model for worker interaction that covers the Mechanical Turk (the SCRAP model) and a data model that demonstrates the diversity of data collected by Turkalytics. We show that Turkalytics is effective at data collection, in spite of it being unobtrusive. Lastly, we describe worker locations, browser environments, activity information, and other examples of data collected by our tool.

Categories and Subject Descriptors

H.5.m [**Information interfaces and presentation (e.g., HCI)**]: [Miscellaneous]

General Terms

Human factors

Keywords

human processing, analytics, mechanical turk, human programming, human computation, crowdsourcing

1. INTRODUCTION

Human computation has been quietly revolutionizing how work in computer science is done. Crowdsourcing marketplaces like Mechanical Turk can provide the labeled data that we have become increasingly dependent on for supervised learning and other tasks. These marketplaces can also provide subjects for user studies rapidly and at low cost [13]. In our department, researchers in visualization, human-computer interaction, computer vision, and natural language processing are all currently using the Mechanical Turk. In industry, Mechanical Turk is now commonly used for tasks like search [9] (e.g., for evaluation or learning to rank) and product categorization.

However, at this early stage, there is a tremendous amount of duplicated effort. Mechanical Turk and similar marketplaces provide a source of labor, but largely avoid the difficult questions of pricing, quality, and reputation. As a result, each requester or system writer using the Mechanical Turk independently identifies appropriate pricing, good and bad work, and good and bad workers. CrowdFlower [1], SmartSheet [2], ATTi's QDEx [8], and TurKit [14] all solve (or fail to solve) these problems independently, for example.

WWW 2011, March 28–April 1, 2011, Hyderabad, India.
ACM 978-1-4503-0632-4/11/03.

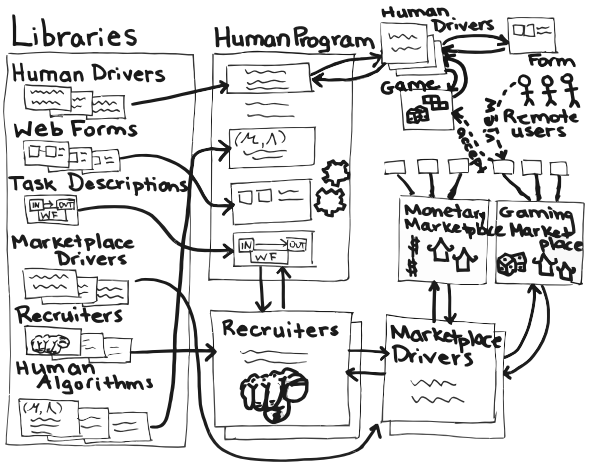

Figure 1: The Human Processing model, figure excerpted from Heymann and Garcia-Molina [10].

In recent work [10], we address this duplicated effort by proposing a model for encouraging code reuse and data sharing. That model (the "Human Processing Model") divides the work of human computation into three primary areas: *human programs*, *human drivers*, and *recruiters* (see Figure 1). A human program is simply a computer program dependent on humans (*workers*) to complete its work. A human program interfaces with workers through human drivers that create web interfaces and otherwise handle worker interaction. Lastly, recruiters are libraries or daemons in charge of posting and pricing available web interfaces to marketplaces like the Mechanical Turk. Thus, the human processing model separates concerns into sub-programs for interacting with workers (human drivers), sub-programs for posting and pricing work (recruiters), and everything else (human programs). Because human drivers and recruiters are often pre-written, this saves authors of human programs from having to re-solve these difficult problems.

One challenge in the human processing model is the collection of reliable data about the workers and the tasks they are performing. This data is needed by our recruiter in particular, but is also needed by *any* system trying to make human

processing more effective: If a task is not being completed, is it because no workers are seeing it? Is it because the task is currently being offered at too low a price? How does the task completion time break down? Do workers spend more time previewing tasks (see below) or doing them? Do they take long breaks? Which are the more "reliable" workers?

This paper addresses the problem of analytics for recruiting workers and studying the performance of ongoing tasks. We describe our prototype system for gathering analytics, illustrate its use, and give some initial findings on observable worker behavior. We believe our tool for analytics, "Turkalytics," is the first human computation analytics tool to be embeddable across human computation systems (see Section 2 for the explicit definition of this and other terms). Turkalytics makes analytics orthogonal to overall system design and encourages data sharing. Turkalytics can be used in stand-alone mode by anyone, without need for our full human-processing infrastructure (Figure 1). Turkalytics functions similarly to tools like Google Analytics [3], but with a different set of tradeoffs (see Section 3.4).

We proceed as follows. Section 2 defines terms and describes the interaction and data models underlying our system. We describe the implementation of Turkalytics based on these models in Section 3. Section 4 describes how a requester uses our system. Sections 5, 6, and 7 present results. Section 5 describes the workload we experienced and shows our architecture to be robust. Section 6 gives some initial findings about workers and their environments. Section 7 considers higher granularity activity data and worker marketplace interactions. Section 8 summarizes related work, and we conclude in Section 9.

2. PRELIMINARIES

We define *crowdsourcing* to be getting one or more remote Internet users to perform work via a marketplace. We call the people doing the work *workers*. We call the people who need the work completed *requesters*. A *marketplace* is a web site connecting workers to requesters, allowing workers to complete tasks for a monetary, virtual, or emotional reward.

In our case, a task is usually a *microtask*, a unit of work which can be done in five minutes or less. Tasks are grouped in *task groups*, so that workers can find similar tasks. Mechanical Turk, the marketplace for which our Turkalytics tool is designed, calls tasks *HITs* and task groups *HITTypes*. When a worker completes a task, we call the completed $(task, worker)$ pair an *assignment* or *work*.

Tasks are *posted* to marketplaces programmatically by requesters using interfaces provided by the marketplaces. A requester usually builds a program called a *human computation system* to ease posting many tasks. (We use "system" in both this specific sense and in a colloquial sense, though we try to be explicit where possible.) The system needs to solve problems like determining when to post, how to price tasks, and how to determine quality work. The human computation system may be based on a framework designed and/or implemented by someone else to solve some of these tasks, like the human processing model [10]. The human computation system may also leave certain problems to outside services, such as our analytics tool (for analytics) or a full service posting and pricing tool like CrowdFlower [1].

The rest of this section describes two models at the core of our Turkalytics tool. The worker interaction model (Section 2.1) represents the steps taken to perform work. The data model (Section 2.2) represents collected data. Our interaction model helps us present results about worker behavior (Section 7) while our data model helps us describe the implementation (Section 3) and requester usage (Section 4).

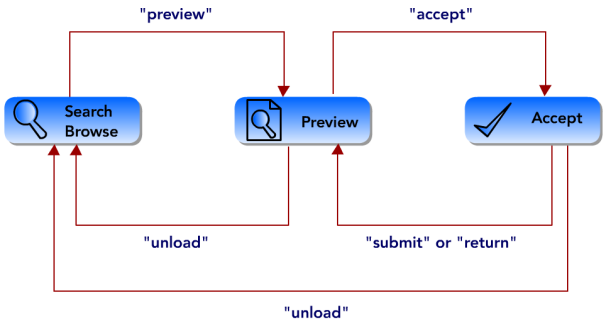

Figure 2: Search-Preview-Accept (SPA) model.

2.1 Interaction Model

Some crowdsourcing marketplaces focus on areas of expertise (e.g., programming or graphic design) while others are more defined by the time span of tasks (e.g., one minute microtasks or month long research projects). Different marketplaces call for different interactions. For example, marketplaces with longer, more skilled tasks tend to have contests or bidding based on proposals, while marketplaces for microtasks tend to have a simpler accept or reject style. Section 2.1.1 describes a simple microtask model, and Section 2.1.2 extends it to cover Mechanical Turk.

2.1.1 Simple Model

The Search-Preview-Accept (SPA) model is a simple model for microtasks (Figure 2). Workers initially are in the *Search* or *Browse* state, looking for work they can do at an appropriate price. Workers can then indicate some interest in a task by entering the *Preview* state through a preview action. *Preview* differs from *Search* or *Browse* in that the worker may have a complete view of the task, rather than some summary information. From *Preview*, the worker can enter the *Accept* state by accepting and actually complete the task. Lastly, the worker can always return to a previous state, for example, a worker can return an accepted task, or leave behind a task that he found uninteresting on preview.

The SPA model fits microtasks well because the overhead of a more complex process like an auction seems to be much too high for tasks that may only pay a few pennies. However, the SPA model does provide flexibility to allow workers to self select for particular tasks and to back out of tasks that they feel unsuited for. The *Accept* state also allows greater control over how many workers may complete a given task, because workers may be prevented from accepting a task.

2.1.2 Mechanical Turk Extensions

Mechanical Turk uses a more complex model than SPA which we call the Search-Continue-RapidAccept-Accept-Preview (SCRAP) model (Figure 3). This model is similar to SPA, but adds two new states, *Continue* and *RapidAccept*. *Continue* allows a worker to continue completing a task that was accepted but not submitted or returned. *RapidAccept* allows a worker to accept the next task in a task group without previewing it first. In practice, the actual states

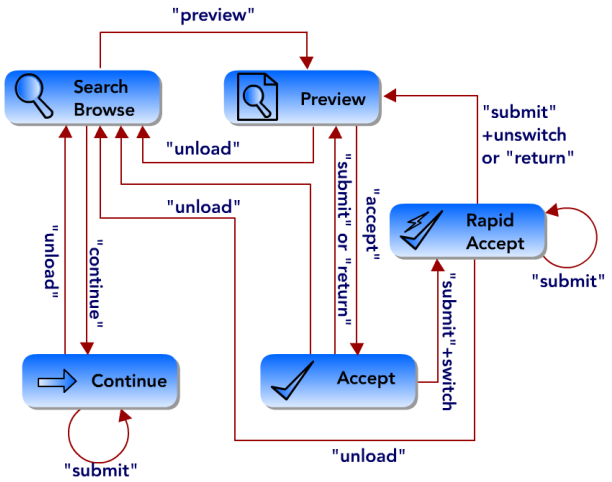

Figure 3: Search-Continue-RapidAccept-Accept-Preview (SCRAP) model.

and transitions in Mechanical Turk are much messier than Figure 3. However, we will see in Section 7.1 that mapping from Mechanical Turk to SCRAP is usually straightforward.

SCRAP is a reasonable model of Mechanical Turk worker activity, but is incomplete in two ways. First, it ignores certain specialized Mechanical Turk features like qualifications. This is primarily because Turkalytics, as an unobtrusive third-party add-on cannot really observe these states. Second, SCRAP describes a particular granularity of activity. As we will see, Turkalytics actually includes data within states, for example, form filling activity or mouse movement. We think of such data as being attached to a state, which is more or less the representation in our data model.

2.2 Data Model

This section uses the terminology of data warehousing and online analytical processing (OLAP) systems. Data in Turkalytics is organized in a star schema, centered around a single fact table, *Page Views*. Each entry in *Page Views* represents one worker visiting one web page, in any of the states of Figure 3. There are a number of dimension tables, which can be loosely divided into task, remote user, and activity tables. The three task tables are:

1. *Tasks:* The task corresponding to a given page view.
2. *Task Groups:* The task group containing a given task.
3. *Owners:* The owner or requester of a given task group.

The four remote user tables are:

1. *IPs:* The IP address and geolocation information associated with a remote user who triggered a page view.
2. *Cookies:* The cookie associated with a given page view.
3. *Browser Details:* The details of a remote user's browser, like *user agent* (a browser identifier like `Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US) AppleWebKit/533.4 (KHTML, like Gecko) Chrome/5.0.375.99 Safari/533.4,gzip(gfe)`) and available plugins (e.g., Flash).
4. *Workers:* The worker information associated with a given remote user.

The two activity tables are:

1. *Activity Signatures:* Details of what activity (and inactivity) occurred during a page view.
2. *Form Contents:* The contents of forms on the page over the course of a page view.

Figure 4 shows an Entity/Relationship diagram. Entities in Figure 4 (the rectangles) correspond to actual tables in our database, with the exception of "Remote Users." Entities attached to "Remote Users" are dimension tables for "Page Views." The circles in the figure represent the attributes or properties of each entity.

There is one set of tables that we have left out for the purpose of clarity. As we will see in Section 3, we need to build up information about a single page view through many separate logging events. As a result, there are a number of tables, which we do not enumerate here, that enable us to incrementally build from logging events into complete logging messages, and then finally into higher level entities like overall activity signatures and page views.

3. IMPLEMENTATION

Turkalytics is implemented in three parts: client-side JavaScript code (Section 3.1), a log server (Section 3.2), and an analysis server (Section 3.3). Section 3.4 gives a broad overview of the design choices we made and limitations of our design.

3.1 Client-Side JavaScript

A requester on Mechanical Turk usually creates a HIT (task) based on a URL. The URL corresponds to an HTML page with a form that the worker completes. Requesters add a small snippet of HTML to their HTML page to embed Turkalytics (see Section 4.1). This HTML in turn includes JavaScript code (`ta.js`) which tracks details about workers as they complete the HIT.

The `ta.js` code has two main responsibilities:

1. *Monitoring*: Detect relevant worker data and actions.
2. *Sending*: Log events by making image requests to our log server (Section 3.2).

`ta.js` monitors the following:

1. *Client Information*: Worker's screen resolution? What plugins are supported? Can `ta.js` set cookies?
2. *DOM Events*: Over the course of a page view, the browser emits various events. `ta.js` detects the `load`, `submit`, `beforeunload`, and `unload` events.
3. *Activity*: `ta.js` listens on a second by second basis for the `mousemove`, `scroll` and `keydown` events to determine if the worker is active or inactive. `ta.js` then produces an activity signature, e.g., `iaaia` represents three seconds of activity and two seconds of inactivity.
4. *Form Contents*: `ta.js` examines forms on the page and their contents. In particular, `ta.js` logs initial form contents, incremental updates, and final state.

`ta.js` sends monitored data to the log server via image requests. We define a *logging event* (or event, where the meaning is clear) to be a single image request. Image requests are necessary to circumvent the same origin policies common in most mainstream browsers, which block actions like sending data to external sites. Special care is also needed to send these image requests in less than two kilobytes due to restrictions in Microsoft Internet Explorer (MSIE). We define a *logging message* to be a single piece of logged data split across one or more events in order to satisfy MSIE's URL size requirements. For example, logging messages sent by `ta.js` include activity signatures, related URLs, client details, and so on (Listing 1 is one such logging message).

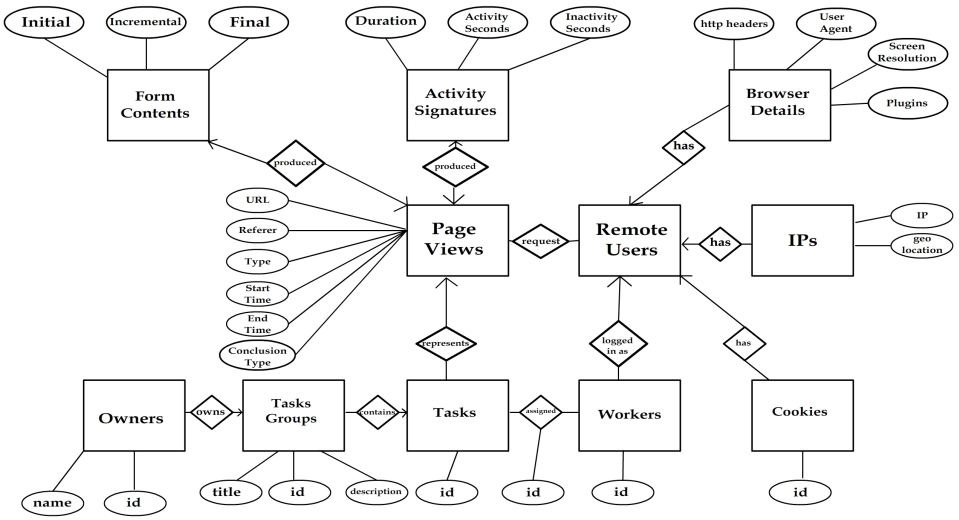

Figure 4: Turkalytics data model (Entity/Relationship diagram).

Note that while we do not discuss the question of privacy in this paper, we gather only data already available to requesters. A single page view can lead to as few as seven or as many as hundreds of image requests. (For example, the NER task described at the beginning of Section 5 can lead to over one hundred requests as it sends details of its form contents because it has over 2,000 form elements.)

3.2 Log Server

The log server is an extremely simple web application built on Google's App Engine. It receives logging events from clients running `ta.js` and saves them to a data store. In addition to saving the events themselves, the log server also records HTTP data like IP address, user agent, and referer. (We intentionally continue the historical convention of misspelling "referer" when used in the context of the HTTP referer, and also do so when referring to the JavaScript document referrer.) A script on the analysis server (Section 3.3) periodically polls the web application, downloading and deleting any new events that have been received. This simplicity pays off: our log server has scaled to over one hundred thousand requests per day.

3.3 Analysis Server

The analysis server periodically polls the log server for new events. These events are then inserted into a PostgreSQL database, where they are processed by a network of triggers. These triggers are arguably the most complex part of the Turkalytics implementation, for four main reasons:

1. *Time Constraints:* One of our goals is for the analysis server to be updated, and query-able, in real-time. Currently, the turnaround from client to availability in the analysis server is less than one minute.
2. *Dependencies:* What to do when an event is inserted into the analysis server may depend on one or more other events that may not have even been received yet.
3. *Incomplete Input:* When Turkalytics has not yet received all logging events pertaining to a message, page view, or any other entity from Figure 4, we call that entity *incomplete*. Nonetheless, requesters should be able to query as much information as possible, even if some entities are incomplete. In fact, many entities will remain incomplete forever. (This is one negative result of an explicit design decision in Section 3.4.)
4. *Unknown Input:* The analysis server may receive unexpected input that conflicts with our model of how the Mechanical Turk works, yet it must still handle this input.

These challenges are sufficiently difficult that our current PL/Python trigger solution is our second or third attempt at a solution. (One earlier attempt made use of dependency handling from a build tool, for example.)

We lack the space to fully describe our triggers here, so we give an example of the functionality instead. Suppose a worker `A19...` has just finished previewing a task `152...`, and chooses to accept it. When the worker loads a new page

```
{ ...
  "HTTP_REFERER":
    "...?assignmentId=1D9...
        &hitId=152...
        &workerId=A1Y9...",
  "PATH_INFO": "/event/relatedUrls",
  "QUERY_STRING":
    "turkaMsgId=2
    &documentReferrerEsc=https%3A%2F...
       %26prevRequester%3DStanford%2B...
       %26requesterId%3DA2IP5GMJBH7TXJ
       %26prevReward%3DUSD0.01...
       %26groupId%3D1ZSQ...
    &turkaConcatNum=0
    &turkaConcatLen=1
    &targetId=f68daad1
    &timeMillis=127...
    &pageSessionId=0.828...
    &clientHash=150...",
... }
```

Listing 1: Excerpt from a related URLs logging event formatted as JSON.

corresponding to the accept state, `ta.js` sends a number of events. Listing 1 shows one such event, a related URLs event detailing the page which referred the worker to the current page. (Our implementation uses JavaScript Object Notation (JSON) as the format for logging events.)

From the `HTTP_REFERER`, Turkalytics can now learn the identifiers for the assignment (`1D9...`), HIT (`152...`), and worker (`A191...`). From the `PATH_INFO`, Turkalytics can learn what type of data `ta.js` is sending (`relatedUrls`). From the `QUERY_STRING`, Turkalytics gets the actual data being sent by `ta.js`, in this case, an escaped referer URL (`documentReferrerEsc`) which in turn includes the reward (`USD0.01`), group identifier (`1ZSQ...`), and other information. The `QUERY_STRING` also includes a `pageSessionId`, which is a unique identifier shared by all events sent as a result of a single page view. (`pageSessionId` is the key for the "Page Views" table.) Note that neither the `HTTP_REFERER` nor the referer sent by `ta.js` as `documentReferrerEsc` represents the worker's previously visited page. The `HTTP_REFERER` seen by our log server is the HIT URL that the worker is currently viewing, and `documentReferrerEsc` is the Mechanical Turk URL containing that HIT URL in an `IFRAME`.

When the event from Listing 1 is inserted into the database, the following functionality is triggered (and more!):

1. The current page view, as specified by `pageSessionId`, is updated to have assignment `1D9...`, HIT `152...`, and worker `A191...`.
2. Page views lacking a worker identifier may have one inferred based on the current page view's worker identifier. (Inference uses an invariant based on distance in time between page views.) For example, the page view associated with the worker's previous preview state is updated to have a worker identifier of `A191...`. (When the worker was in the preview state, the assignment and worker identifiers were unknown, but now we can infer them based on this later information.)
3. If not already known, a new task group `1ZSQ...` with a reward of one cent is added.
4. If not already known, a new mapping from the current HIT `152...` to the task group `1ZSQ...` is added.
5. If not already known, the requester name and identifier are added to the task group and owner entities.

This example shows that incrementally building entities from Figure 4 in real-time requires careful consideration of both event dependencies and appropriate invariants.

3.4 Design Choices

There are four main considerations in Turkalytics' design:

1. *Ease*: We wanted Turkalytics to be easy for requesters to use and install.
2. *Unobtrusiveness*: We wanted Turkalytics to be as invisible as possible to workers as they perform work, and not to impact the operation of requesters.
3. *Data Collection*: We wanted to gather as much worker task completion data as possible.
4. *Cross-Platform*: We wanted Turkalytics to work across a number of different human computation systems for posting work to Mechanical Turk, because such systems are currently quite heterogeneous.

Our requirements that Turkalytics be easy, unobtrusive, and cross-platform led us to build our tool as embeddable JavaScript, and to use simple cross-platform ideas like sessions and cookies to group events by workers.

It is perhaps worth taking a moment to note why building an analytics tool like Turkalytics, and in particular building it as embeddable, cross-platform JavaScript is nontrivial. First, we do not have direct access to information about the state of Mechanical Turk. We do not want to access the Mechanical Turk API as each of our requesters. However, even if we did, the Mechanical Turk API does not allow us to query fine grained data about worker states, worker activity, or form contents over time. Nor does it tell us which workers are reliable, or how many workers are currently using the system. Second, data collected is often incomplete, as discussed in Section 3.3, and we often need to infer additional data based on information that we have. Third, remote users can change identifiers in a variety of ways, and in many cases we are more interested in the true remote user than any particular worker identifier. All of these challenges, in addition to simply writing JavaScript that works quickly and invisibly across a variety of unknown web browsers with a variety of security restrictions (same origin policy, third party cookies), make writing an analytics tool like Turkalytics difficult.

Two of our design considerations, "unobtrusiveness" and "data collection" are in direct opposition to one another. For example, consider the following trade-offs:

- `ta.js` could send more logging messages with more details about the worker's browser state, but this may be felt through processor, memory, or bandwidth usage.
- `ta.js` could sample workers and only gather data from some of them, improving the average worker's experience, but reducing overall data collection.
- `ta.js` could interfere with the worker's browser to ensure that all logging events are sent and received by our logging server, for example, by delaying submission of forms while logging messages are being sent.

These options represent a spectrum between unobtrusiveness and data collection.

We chose to send fairly complete logging messages and avoid sampling. This is because we believe that workers are more motivated to deal with minor performance degradation (on the order of hundreds of milliseconds) than regular web visitors. This is quite different than the assumptions behind tools like Google Analytics. Nonetheless, we draw the line at interfering with worker behavior, which we deem too obtrusive. A result of this decision is that we may occasionally have incomplete data from missed logging messages.

A current technical limitation of our implementation is a focus on HTML forms. HITs that make use of Flash or an `IFRAME` may produce incomplete activity and form data. However, there is nothing in our design which means that such cases could not be handled eventually.

4. REQUESTER USAGE

Requesters interact with Turkalytics at two points: installation (Section 4.1) and reporting (Section 4.2). Our goal in this section is to illustrate just how easy our current Turkalytics tool is to use currently and to show just how much benefit requesters get. (If the reader is currently writing a Mechanical Turk program or system, do get in touch with the authors about embedding Turkalytics!)

4.1 Installation

In most cases, embedding Turkalytics simply requires adding a snippet of HTML (see Listing 2) to each HTML page corresponding to a posted HIT. (See Section 3.1 for

```
<script type="text/javascript"
        src="https://.../ta.js">
</script>
<script type="text/javascript">
  Turka.Frontend.startAllTracking("...");
</script>
```

Listing 2: Turkalytics embed code.

more on how HTML pages relate to HITs.) Most systems for displaying HITs have some form of templating system in place, so this change usually only requires a copy-and-paste to one file. An important special case of a human computation system with a templating system is the Mechanical Turk requester bulk loading web interface [4]. Underlying that interface is a templating system which generates HTML pages on Amazon's S3 system [5], so Listing 2 works there as well (by adding it to the bottom of each template). These two cases, requesters posting HTML pages and requesters using the bulk interface, cover all of our current requesters.

Listing 2 has two parts elided. The first "..." is where `ta.js` is located, on our server. This does not change across installations. The second "..." is an identifier identifying the particular requester. Currently, we assign each Turkalytics requester a hexadecimal identifier, like `7e3f6604`. Once the requester has added the snippet from Listing 2 with these changes, they are done. (Requesters lacking SSL also need to use a simple workaround script due to referer handling, but such requesters are rare.) In our experience, the process usually takes less than five minutes and is largely invisible to the requester afterwards.

Two implementation notes bear pointing out here about the hexadecimal identifier. The first note is that due to the web browser context, and due to our status as a third party, it is possible that an "attacker" could send our system fake data. At this stage, there is not a lot of reason to do this, and this is a problem with most analytics systems. The second note is that the hexadecimal identifier allows us to easily partition our data on a per requester basis. Our current analysis server uses a multitenant database where we query individual requester statistics using this identifier, but could easily be split across multiple databases.

4.2 Reporting

Once Turkalytics is installed (Section 4.1), all that remains is to later report to the requester what analysis we have done. Like most data warehousing systems, we have two ways of doing this. We support ad hoc PostgreSQL queries in SQL and we are in the process of implementing a simple web reporting system with some of the more common queries. In fact, most of the data in this paper was queried from our live system, including Tables 1 and 2, Figures 6 and 7, and most of the inline statistics. (The only notable exception is Figure 5, where it is somewhat awkward to compute sequential transitions in SQL.)

To give a flavor for what you can do, Listing 3 gives two example queries that run on our real system. For example, suppose we want to know which requesters in our system are the heaviest users of Mechanical Turk. The first query computes total number of tasks and total payout aggregated by requester by joining page view data with task group data. An

```
SELECT tg.requester_name
     , sum(tg.reward_cents) AS total_cents
     , count(*) AS num_submits
FROM page_views AS pv
     , task_groups AS tg
WHERE pv.task_group_id=tg.task_group_id
  AND pv.page_view_type='accept'
  AND pv.page_view_end='submit'
GROUP BY tg.requester_name;

SELECT tg.requester_name
     , pv.task_group_id
     , sum(tg.reward_cents * 3600)
         / sum(a.active_secs)
     , sum(tg.reward_cents * 3600)
         / sum(a.total_secs)
FROM page_views AS pv
   , task_groups AS tg
   , activity_signatures AS a
WHERE pv.task_group_id=tg.task_group_id
  AND pv.page_view_id=a.page_view_id
  AND pv.page_view_type='accept'
  AND pv.page_view_end='submit'
  AND a.is_complete
GROUP BY tg.requester_name
       , pv.task_group_id;
```

Listing 3: Two SQL reporting queries.

example output tuple is `("Petros Venetis", 740, 160)`. That output tuple means that the requester `Petros Venetis` spent $7.40 USD on 160 tasks. The second query computes the hourly rate of workers based on active and total seconds grouped by task group. (The query does so by joining page views, task groups, and activity data, and using the activity data to determine amount of time worked.) We might want to do this, for example, to determine appropriate pricing of a future task based on an estimate of how long it takes. An example output tuple is `("Paul H", "1C4...", 6101.695, 122.553)`. That output tuple means that for task group `1C4...` owned by `Paul H`, the hourly rate of workers based on the number of active seconds was ≈$61.02 USD, while the hourly rate based on total time to completion was ≈$1.23 USD. (Note that the example tuple has a very large discrepancy between active and total hourly rate, because the task required workers to upload an image created offline.)

5. RESULTS: SYSTEM ASPECTS

This section, and the two that follow (Sections 6 and 7) describe our production experience with the Turkalytics system. Our data for these sections is collected over the course of about a month and a half starting on June 14th, 2010. The data consists of 12,370 tasks, 125 worker days, and a total cost of $543.66. In our discussion below, we refer to three groups of tasks posted by requesters using our tool:

1. *Named Entity Recognition (NER):* This task, posted in groups of 200 by a researcher in Natural Language Processing, asks workers to label words in a Wikipedia article if they correspond to people, organizations, locations, or demonyms. (2,000 HITs, 1 HITType, more than 500 workers.)

2. *Turker Count (TC):* This task, posted once a week by a professor of business at U.C. Berkeley, asks workers to push a button, and is designed just to gauge how many workers are present in the marketplace. (2 HITs, 1 HITType, more than 1,000 workers each.)
3. *Create Diagram (CD):* This task, posted by the authors, asked workers to draw diagrams for this paper based on hand drawn sketches. In particular, Figures 2, 3, and 5 were created by worker `A1K17L5K2RL3V5` while Figure 4 was created by worker `ABDDE4BOU86A8`. ($\approx$ 5 HITs, 1 HITType, more than 100 workers.)

There are two questions worth asking about our Turkalytics tool itself. The first is whether the system is performant, i.e., how fast it is and how much load it can handle. The second is whether it is successfully collecting the intended monitored information. (Because Turkalytics is designed to be unobtrusive, there are numerous situations in which Turkalytics might voluntarily lose data in the interest of a better client experience.) This section answers these questions focusing on the client (Section 5.1), the logging server (Section 5.2), and the analysis server (Section 5.3).

5.1 Client

Does ta.js effectively send remote logging messages?

We asked some of our requesters to add an image tag corresponding to a one pixel GIF directly above the Listing 2 HTML in their HITs. Based on access to this "baseline" image, we can determine how many remote users viewed a page versus how many actually loaded and ran our JavaScript. (This assumes that remote users did not block our server, and that they waited for pages to load.)

Overall, the baseline image was inserted for 25,744 URLs. Turkalytics received JavaScript logging messages for all but 88 of those URLs, which means that our loss rate was less than 0.5%. There is no apparent pattern in which messages are missing on a per browser or other basis. Our `ta.js` runs on all modern browsers, though some features vary in availability based on the browser. (For example, Safari makes it difficult to set third party cookies and early versions of MSIE slow down form contents discovery due to DOM speed.)

How complete is activity sending?

Activity data is sent periodically as logging messages by `ta.js`. However, as with other logging messages, the browser thinks we are loading a series of images from the logging server rather than sending messages. As a result, if the worker navigates away from the page, the browser may not bother to finish loading the images. After all, there is no longer a page for the images to be displayed on!

How commonly are activity logging messages lost? We looked at activity signatures for 9,884 page views corresponding to NER tasks. Each page view was an accepted task which the worker later submitted. We computed an *expected number of activity seconds* for a given page view by subtracting the timestamp of the first logging event received by Turkalytics from the timestamp of the last logging event. If we have within 20 seconds, or within 10% of the total expected number fo activity seconds, whichever is less, we say that we have full activity data. (Activity monitoring may take time to start, so we leave a buffer before expecting activity logging messages.) For 8,426 of these page views, or about 85%, we have "full" activity data in this sense.

How fast and correct is form content sending?

Checking the form contents to send to the logging server usually takes on the order of a few hundred milliseconds every ten to thirty seconds. This assumes a modern browser and a reasonably modern desktop machine. Of the 9,884 NER page views accepted and submitted from the previous section, 8,049 had complete form data.

5.2 Logging Server

Given the simplicity of the logging server, it only makes sense to ask what the peak load has been and whether there were any failed requests. (Failed requests are logged for us by the App Engine.) In general, Mechanical Turk traffic is extremely bursty—at the point of initial post, many workers complete a task, but then traffic falls off sharply (see Section 7.2). However, our architecture handles this gracefully. In practice, we saw a peak requests/second of about ten, and a peak requests/day of over 100,000, depending on what tasks were being posted by our requesters on a given day. However, there is no reason to think that these are anywhere near the limits of the logging server. During the period of our data gathering, we logged 1,659,944 logging events, and we lost about 20 per day on average due to (relatively short) outages in Google's App Engine itself.

5.3 Analysis Server

Our analysis server is an Intel Q6600 with four gigabytes of RAM and four regular SATA hard drives located at Stanford. We batch loaded 1,515,644 JSON logging events in about 520 minutes to test our trigger system's loading speed. Despite the fact that our code is currently single threaded and limited to running forty seconds of every minute, our batch load represents an amortized rate of about 48 logging events per second. The current JSON data itself is about 2.1 gigabytes compressed, and our generated database is about 4.6 gigabytes on disk. Both the data format and the speed of insertion into the analysis server could both be optimized: currently the insertion is mostly CPU bound by Python, most likely due to JSON parsing overhead.

6. RESULTS: WORKER ASPECTS

Where are workers located?

Most demographic information that is known about Mechanical Turk workers is the result of surveys on the Mechanical Turk itself. Surveys are necessary because the workers themselves are kept anonymous by the Mechanical Turk. However, such surveys can easily be biased, as workers appear to specialize in particular types of work, and one common specialization is filling out surveys.

Turkalytics allows us to test the geographic accuracy of these past surveys. We use the "GeoLite City" database from MaxMind to geolocate all remote users by IP address. (MaxMind claims this database is 99.5% accurate at the country level [6].) Results are shown in Table 1. For example, the first line of Table 1 shows that in our data, the United States represented 2,534 unique IP addresses (29.84% of the total), 1,299 unique workers (44.716% of the total), 199 of the unique workers who did the NER task, and 1,011 of the unique US workers who completed the TC task.

There are two groups of countries in the data. The United States and India are the first group, and they represent

	Overall				Named Entity	Turk Count
	# IPs	% IPs	# Workers	% Workers	# Workers	# Workers
United States	2534	29.840	1299	44.716	199	1011
India	4794	56.453	1116	38.417	227	717
Philippines	127	1.496	52	1.790	15	32
United Kingdom	92	1.083	43	1.480	11	27
Canada	86	1.013	42	1.446	10	33
Germany	50	0.589	16	0.551	6	10
Australia	32	0.377	16	0.551	4	10
Pakistan	49	0.577	15	0.516	5	10
Romania	96	1.130	14	0.482	5	7
Anonymous Proxy	13	0.153	12	0.413	0	13

Table 1: Top Ten Countries of Turkers (by Number of Workers). 2,884 Workers, 8,216 IPs total.

about 80% of workers. The second group is everywhere else, consisting of about 80 other countries, and 20% of the workers. This second group is more or less power law distributed. We suspect that worker countries are heavily biased by the availability of payment methods. Mechanical Turk has very natural payment methods in the United States and India, but not elsewhere (e.g., even in other English speaking countries like Canada, the United Kingdom, and Australia).

Comparing the NER and TC tasks, we can see that Indians are more prevalent on the NER task. However, regardless of grouping, the nationality orderings seem to be fairly similar, with the caveat that Indians have many more IPs than Americans. This suggests that previous survey data may be slightly biased based on respondents, but overall may not be terribly different from the true underlying worker demographics.

What does a "standard" browser look like?

The most common worker screen resolutions are 1024x768 (2266 workers at at least one point), 1280x800 (1166 workers), 1366x768 (670 workers), 1440x900 (494 workers), 1280x1024 (451 workers), and 800x600 (228 workers). No other resolution has more than 200 workers. Given an approximate browser height of 170px and a Mechanical Turk interface height of 230px or more, a huge number of workers are previewing (and possibly completing) tasks in less than 400px of screen height. As a caveat, some workers may be double counted as they switch computers or resolutions. The average is about 1.5 distinct resolutions per worker, so most workers have one or two distinct resolutions.

About half of our page views are by Firefox users, and about a quarter are by MSIE users. In terms of plugins, Java and Flash represent 70–75% of our page views, each, while PDF and WMA represent 50–55% each. These may be underestimates based on our detection mechanism (navigator MIME types).

Are workers identifiable? Do they switch browsers?

It is becoming increasingly common to use the Mechanical Turk as a subject pool for research studies. A growing body of literature has looked at how to design studies around the constraints of Mechanical Turk. One key question is how to identify a remote user uniquely. For example, how do I know that an account for a 30 year old woman from Kansas is not really owned by a professional survey completer with a number of accounts in different demographic categories?

One solution is to look at reasonably unique data associated with a remote user. Table 2 shows the number of user agents, IP addresses, and identifier cookies associated with a given worker. Ideally, for identification purposes, each of these numbers would be one. In practice, these possibly unique pieces of data seem to vary heavily by worker. Common user agent strings, dynamic IPs, and downgrading or blocking of cookies (particularly third party cookies as Turkalytics uses) are all possible reasons for this variability. For example, the worker three from the bottom had 84 distinct IP addresses over the course of 226 page views, but nonetheless kept the same two tracking cookies throughout. On the other hand, the first worker in the table had 17 tracking cookies over 619 page views, but only had one IP address throughout. On average, for active workers, the user agent to page view ratio is about 1:25, the IP to page view ratio is about 1:10, and the cookie to page view ratio is about 1:11. These numbers are skewed by special cases however, and the median numbers are usually lower. ("Cheaters" appear rare, though one remote user with a single cookie seems to have logged in seven different times, as seven different workers, to complete the TC task.)

Worker	Counts				Country
	UAs	IPs	Cookies	Views	
AXF...	3	1	17	619	India
A1B...	2	9	4	618	Multinational
A1K...	5	23	8	537	India
A3O...	4	13	68	502	India
A2C...	4	47	33	462	India
A3I...	4	2	3	450	United States
A2I...	3	4	2	393	United States
A1V...	4	14	1	314	United States
A1C...	4	10	7	303	India
A31...	3	11	2	288	India
A2H...	8	6	8	268	India
A29...	1	17	2	244	India
A3J...	3	84	2	226	India
A2O...	3	3	4	225	United States
A1E...	1	25	5	225	India

Table 2: The number of user agents, IP addresses, cookies and views for top workers by page views.

7. RESULTS: ACTIVITY ASPECTS

Section 2.1 gave a model for interaction with Mechanical Turk. This section looks at what behavior that model produces. Section 7.1 looks at what states and actions occur

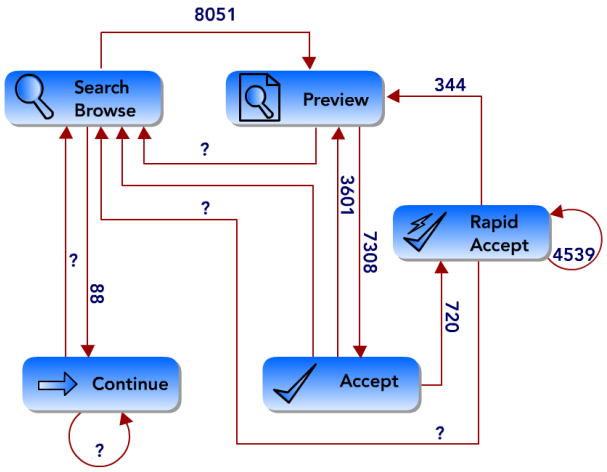

Figure 5: Number of transitions between different states in our dataset. Note: These numbers are approximate and unload states are unlabeled.

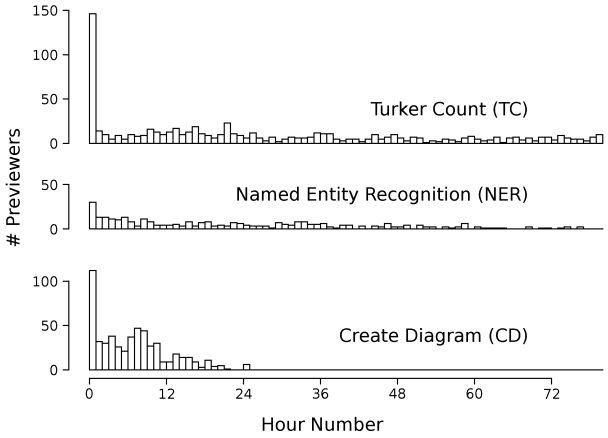

Figure 6: Number of new previewers visiting three task groups over time.

in practice. Section 7.2 looks in particular at previewing interactions. Section 7.3 looks at activity data generated by workers within page views. Our main goals are to show that Turkalytics is capable of gathering interesting system interaction data and to illustrate the tradeoffs of Mechanical Turk-like (i.e., SCRAP-like) systems.

7.1 What States/Actions Occur in Practice?

Workers in the SCRAP model can theoretically execute a large number of actions. However, we found that most transitions were relatively rare. Figure 5 shows the transitions between states that we observed. (Very rare, unclear, or "unload" transitions are marked with question marks.) For example, we observed 720 transitions by workers from *Accept* to *RapidAccept*, and 344 transitions out of the *RapidAccept* state. To generate Figure 5, we assume the model described in Section 2.1 and that workers are "single threaded," that is, they use a single browser with a single window and no tabs. These assumptions let us infer state transitions based on timestamp, which is necessary because of the referer setup described in Section 3.3. Over 88% of our observed state transitions are transitions in our mapped SCRAP model, so our simplifying assumptions appear relatively safe.

Do workers use the extensions provided by the SCRAP model above and beyond the SPA model? We found that *RapidAccept* was used quite commonly, but *Continue* was quite rare. About half of the workers who chose to do large numbers of tasks chose to *RapidAccept* often rather than continuously moving between the *Preview* and *Accept* state. However, continues represent less than 0.5% of our action data, and returns about 2%. (We suspect that the transition to *Accept* from *Preview* is particularly common in our data due to the prevalence of the simple Turker Count task.)

7.2 When Do Previews Occur?

A common Mechanical Turk complaint is that the interfaces for searching and browsing constrain workers. In particular, Chilton et al. observe that workers primarily sort task groups by how recently they were created and how many tasks are available in the group. This observation appears to be true in our data as well.

Figure 6 shows when previously unseen workers preview the NER, TC, and CD task groups immediately after an instance was posted. For example, in the first hour of availability of the TC task, almost 150 workers completed the task. The NER task group has many tasks, but is posted only once. The TC task group has only one task, and is only posted once. The CD task group has five tasks, but is artificially kept near the top of the recently created list. In Figure 6, both NER and TC show a stark drop off in previews after the first hour when they leave the most recently created list. NER drops off less, likely because it is near the top of the tasks available list. CD drops off less than TC, suggesting artificial recency helps. These examples suggest that researchers should be quite careful when drawing conclusions about worker interaction (e.g., due to pricing) because the effect due to rankings is quite strong.

7.3 Does Activity Help?

Turkalytics collects activity and inactivity information, but is this information more useful than lower granularity information like the duration that it took for the worker to submit the task? It turns out that there are actually two answers to this question. The first answer is that, as one might expect, the amount of time a worker is active is highly correlated with the total amount of time a worker spends completing the task in general. The second answer is that, despite this, signature data does seem to clarify the way in which a worker is completing a particular task.

We looked at the activity signatures of 321 workers who had at least one complete signature and had completed the NER task. The Pearson correlation between the number of active seconds and the total number of seconds for these workers was 0.88 (see Figure 7). However, the activity signatures do give a more granular picture of the work style of different workers. Figure 8 shows two quite different activity signatures, both of which end in completing an accepted task. The first signature shows a long period of inactive seconds (`i`) followed by bursts of active seconds (`a`), while the second signature shows a short period of mostly active seconds. One might prefer one work completion style or the other for particular tasks.

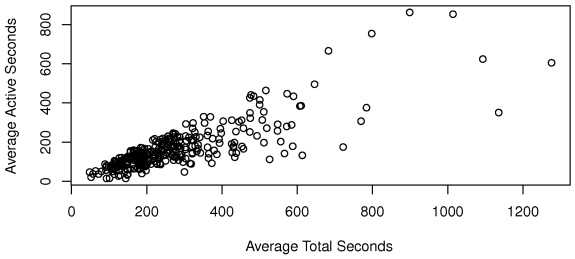

Figure 7: Plot of average active and total seconds for each worker who completed the NER task.

```
diiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii
.............. 300 inactive seconds .............
iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii
iaaaaaaiiiiaaaaaaaiaiaaaaaaaaaaaaaaaaiaaiaaaaaaaaaa
aaaaaaaaaaaaaaaiiiiiiiiiiaaaaaaaaaaaaaaaiaaaiiiaaaa
aaaaaiiiiiiiiaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaiaaaiaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaiiiiiiiiiiiiiiiaiaiiiiiiiaaiiaiaaaaaaaaiiaaaaaa
aaiiaiiaaaaaaaaaiaaaaaaaiiiiaaiaaaaaaaasbu
```

(a) 688 Second Activity Signature

```
daaaaiaaaaaiaaaaaaaaaaaaiiiiiiaaaaaaaiiiaaaaaaaaaa
aaaaiiiiaaiiiaaaiiiiaaaaaiaaaaiaaaaaiiaaaaiiiasabu
```

(b) 96 Second Activity Signature

Figure 8: Two activity signatures showing different profiles for completing a task. Key: a=activity, i=inactivity, d=DOM load, s=submit, b=beforeunload, u=unload.

8. RELATED WORK

Our system, and the results presented, are related to four main areas of work: human computation systems, analytics, Mechanical Turk demographic research, and general Mechanical Turk work. With respect to human computation systems, several were cited in the introduction [1], [2], [8], [14]. Our intent is for our tool to improve such systems and make building them easier. With respect to analytics, numerous analytics tools (e.g., [3]) exist in industry, though there does not appear to be a great deal of work in the academic literature about such tools. With respect to demographics, independent work by Ipeirotis [11], [12] and Ross et al. [15] used worker surveys to illustrate the changing demographics of Mechanical Turk over time. (Section 6 more or less validates these previous results, as well as adding a more recent data point.) With respect to general Mechanical Turk research, the most common focuses to date have been conducting controlled experiments [13] and performing data annotation in areas like natural language processing [16], information retrieval [7], and computer vision [17].

9. CONCLUSION

We presented Turkalytics, a tool for gathering data about workers completing human computation tasks. We envision Turkalytics as part of a broader system, in particular a system implementing the Human Processing model. However, one big advantage of our design for Turkalytics is that it is not tied to any one system. Turkalytics enables both code sharing among systems (systems need not reimplement worker monitoring code) and data sharing among systems (requesters benefit from data gathered from other requesters). Our contributions include interaction and data models, implementation details, and findings about both our system architecture and the popular Mechanical Turk marketplace. We showed that our system was scalable to more than 100,000 requests/day. We also verified previous demographic data about the Turk, and presented some findings about location and interaction that are unique to our tool. Overall, Turkalytics is a novel and practical tool for human computation that has already seen production use.

10. REFERENCES

[1] http://crowdflower.com/.

[2] http://www.smartsheet.com/.

[3] http://www.google.com/analytics/.

[4] http://requester.mturk.com/.

[5] http://s3.amazonaws.com/.

[6] http://www.maxmind.com/app/geolitecity.

[7] O. Alonso and S. Mizzaro. Can we get rid of TREC Assessors? Using Mechanical Turk for Relevance Assessment. In *SIGIR '09 Workshop on the Future of IR Evaluation*.

[8] D. Feng. Talk: Tackling ATTi Business Problems Using Mechanical Turk. Palo Alto Mechanical Turk Meetup, 2010.

[9] J. Galante. CrowdFlower's Virtual Pay for Digital Purchases. http://www.businessweek.com/magazine/content/10_26/b4184041335224.htm.

[10] P. Heymann and H. Garcia-Molina. Human Processing. Technical report, 2010.

[11] P. Ipeirotis. Mechanical Turk: The Demographics. http://behind-the-enemy-lines.blogspot.com/2008/03/mechanical-turk-demographics.html.

[12] P. Ipeirotis. The New Demographics of Mechanical Turk. http://behind-the-enemy-lines.blogspot.com/2010/03/new-demographics-of-mechanical-turk.html.

[13] A. Kittur, E. H. Chi, and B. Suh. Crowdsourcing User Studies with Mechanical Turk. In *CHI '08*.

[14] G. Little, L. Chilton, M. Goldman, and R. Miller. TurKit: Tools for Iterative Tasks on Mechanical Turk. In *HCOMP '09: SIGKDD Workshop on Human Computation*.

[15] J. Ross, L. Irani, M. Silberman, A. Zaldivar, and B. Tomlinson. Who are the Crowdworkers?: Shifting Demographics in Mechanical Turk. In *CHI '10*.

[16] R. Snow, B. O'Connor, D. Jurafsky, and A. Ng. Cheap and Fast—But is it Good?: Evaluating Non-expert Annotations for Natural Language Tasks. In *EMNLP'08*.

[17] A. Sorokin and D. Forsyth. Utility Data Annotation with Amazon Mechanical Turk. In *CVPRW'08*.