Thomas J. Leeper

Crowdsourced data preprocessing with R and Amazon Mechanical Turk

Crowdsourced Data Preprocessing with R and Amazon Mechanical Turk

by Thomas J. Leeper

Abstract This article introduces the use of the Amazon Mechanical Turk (MTurk) crowdsourcing platform as a resource for R users to leverage crowdsourced human intelligence for preprocessing "messy" data into a form easily analyzed within R. The article first describes MTurk and the **MTurkR** package, then outlines how to use **MTurkR** to gather and manage crowdsourced data with MTurk using some of the package's core functionality. Potential applications of **MTurkR** include construction of manually coded training sets, human transcription and translation, manual data scraping from scanned documents, content analysis, image classification, and the completion of online survey questionnaires, among others. As an example of massive data preprocessing, the article describes an image rating task involving 225 crowdsourced workers and more than 5500 images using just three **MTurkR** function calls.

Introduction

Often people use R because it is extensible, robust, and free. It can do many things, but doing those many things generally requires data structures that can be handled computationally. Yet sometimes R users are faced with messy data that are not "R-ready." Examples include: when working with handwritten survey responses, digitized texts that cannot be read by optical character recognition, images, etc. Other times an analyst may face machine-readable data that requires human interpretation to categorize, translate, or code the data, e.g., someone wishing to build an automated classifier needs a human-categorized training set to test their implementation.

In such cases, making the leap from these raw data to R data structures can entail considerable human labor. Such needs for human labor in data preprocessing has provoked interest in online crowdsourcing platforms (Schmidt, 2010; Chen et al., 2011) to bring human intelligence to tasks that cannot be easily accomplished through computation alone. This paper describes the use of **MTurkR** (Leeper, 2016) to leverage the Amazon Mechanical Turk (MTurk) crowdsourcing platform to bring human intelligence into R. The article begins by laying out the need for occasional human intelligence in data preprocessing, then describes MTurk and its vocabulary, and introduces **MTurkR**.

The need for human intelligence

Some data cannot be computationally preprocessed. Other data can be handled computationally only with difficulty. In these cases, data preprocessing can be a time consuming and expensive task because of the human intelligence required. Archetypal needs for this kind of human intelligence include the collection of data which cannot be automated (e.g., unstructured or malformed web data), transcription of files into machine-readable data (e.g., audio, images, or handwritten documents scanned as PDFs), tasks that are laborious to translate from an R-readable but non-computable data structure into a format that can be readily analyzed (e.g., text answers to free-response survey questions), or massive-scale machine readable data that require human interpretation (e.g., the data used in generating a training set for supervised learning algorithms).

Due to the manual nature of these tasks, preprocessing such data can become challenging, especially as the size of the dataset increases. Crowdsourcing these data preprocessing needs is therefore one way to obtain the scalable human intelligence needed to preprocess even very large "messy" datasets. As opposed to an analyst engaged in manual preprocessing, crowdsourcing offers the possibility to leverage multiple sources of human intelligence, in parallel, thereby improving reliability and speed. Amazon Mechanical Turk (MTurk) stands out as one of the largest crowdsourcing platforms currently available and, its powerful API is now accessible directly in R through **MTurkR**.

MTurk core concepts

Amazon Mechanical Turk is a crowdsourcing platform designed by Amazon as part of its suite of Amazon Web Service (AWS) tools to provide human intelligence for tasks that cannot be readily, affordably, or feasibly automated (Amazon.com, 2012). Because MTurk provides the web application for recruiting, paying, and managing human workers, the effort necessary to move a data cleaning task into the cloud is relatively effortless and, with **MTurkR**, can, in large part, be managed directly in

R. While many early adopters of MTurk as a data generation tool have come from computer science (Mason and Suri, 2012; Kittur et al., 2008), more recent attention has also emerged in the social sciences where MTurk's pool of workers are seen as a low-cost participant pool for human subjects research (Buhrmester et al., 2011; Berinsky et al., 2010; Paolacci et al., 2010). This article provides a sufficiently general overview of MTurk and **MTurkR** to enable its use for a variety of purposes, but focuses primarily on the uses of MTurk for data preprocessing.[1]

Key terms

MTurk connects *requesters*, who are willing to pay *workers* to perform a given task or set of tasks at a specified price per task. These "Human Intelligence Tasks" (HITs), are the core element of the MTurk platform. A HIT is a task that a requester would like one or more workers to perform. Every HIT is automatically assigned a unique HITId to identify this HIT in the system. Performance of that HIT by one worker is called an *assignment*, indexed by a unique AssignmentId, such that a given worker can only complete one assignment per HIT but multiple workers can each complete an assignment for each HIT. As a simple example, if a HIT is a PDF file to be transcribed, the researcher might want three workers to complete the transcription in order to validate the effort and therefore make three assignments available for this HIT.

In other situations, however, a researcher may want workers to complete a set of related tasks. For example, the researcher may want to categorize 5000 text statements such as free response answers to a survey question into a set of fixed categories. Each of these statements could be treated as a separate HIT, grouped as a *HITType* with one (or more) assignment(s) available for each HIT. While a worker could complete all 5000 assignments they might also code fewer (e.g., 50 statements), thereby leaving 4950 assignments for other workers to claim.

Workers choose which HITs to complete and how many HITs they want to complete at any given time, depending on their own time, interests, and the payments that requesters offer in exchange for completing an assignment for a given HIT.[2] A requester can offer as low as $0.005 per assignment. Similarly, requesters can pay any higher amount, but that may not be cost-effective given the market forces in play on MTurk. Workers increasingly expect competitive wages, at a rate of at least U.S. minimum hourly wage.

Once a worker completes a HIT, the requester can *review* the assignment – that is, see the responses provided by the worker to the HIT – and the requester can either *approve* (and thus pay the worker the pre-agreed "reward" amount) or *reject* (and not pay the worker).[3] This review process can be relatively automated or handled manually by the requester.

The MTurk system records all workers that have ever performed work for a given requester and provides an array of functionality for tracking, organizing, paying, and corresponding with workers. In particular, the system allows requesters to regulate who can complete HITs through the use of *QualificationRequirements* (e.g., a worker's previous HIT approval rate, their country of residence, or a requester-defined qualification such as past performance or previously evaluated skills).

Sandbox environment

One final point is that MTurk has both a "live" website and development *sandbox*, where the service can be tested without transacting any money. The sandbox can be a useful place to create and test HITs before making them available for workers. Note, however, that the two systems – despite operating with identical code – have separate databases of HITs, HITTypes, qualifications, workers, and assignments so code may not directly translate between sandbox and the live server.

MTurk API and other packages

Amazon provides software development kits for Python, Ruby, etc. as well as a rudimentary command-line utility, but no officially supported client for R. The **MTurkR** package fills this gap, enabling R

[1] Users specifically interested in social science survey and experimental applications should consult Leeper (2013) and the **MTurkR** documentation.

[2] Workers also communicate about the quality of HITs and requesters on fora such as TurkOpticon (`http://turkopticon.differenceengines.com/`), MTurk Forum (`http://mturkforum.com/`), Turker Nation (`http://www.turkernation.com/`), and Reddit pages (`http://www.reddit.com/r/HITsWorthTurkingFor/` and `http://www.reddit.com/r/mturk`).

[3] Note that Amazon also charges a surcharge on all worker payments. Also, if the requester thinks the work merits additional compensation (or perhaps if workers are rewarded for completing multiple HITs of a given HITType), the requester can also pay a *bonus* of any amount to the worker at any point in the future.

users to fully manage an MTurk workflow, from submitting "messy" data to MTurk, reviewing work completed by workers, and retrieving completed work as an R data frame.[4]

The MTurkR package

Before using MTurk, a MTurk requester account is necessary. These which can be created at http://www.mturk.com.[5] It is also helpful from a practical perspective to have a worker account, so that you can test your own HITs interactively and have the requester-worker relationship necessary to test some MTurk features (e.g., contacting workers or setting up qualifications). **MTurkR**'s access to the MTurk API requires Amazon Access Keys, which can be setup at https://console.aws.amazon.com/iam/home?#security_credential. The *keypair* is a linked *Access Key ID* and a *Secret Access Key*.

MTurkR is implemented in a functional programming style, with the core functionality enabling the creation of HITs and retrieval of resulting assignment data. All of this functionality is described here, as well as in detailed examples in the **MTurkR** package documentation (Leeper, 2016). As a web API client, the package provides a complete wrapper for all API features using function names closely mapped onto API endpoints, making it easy to cross-reference MTurk API documentation with **MTurkR** functionality. **MTurkR** performs HTTP requests to the MTurk API using **curl** (Ooms, 2016) and parses responses using **XML** (Temple Lang, 2012). In almost all cases, responses are converted into data frames. In the event an API request fails, error reporting information is returned instead of the standard data structure.[6]

A simple "hello world!" test in **MTurkR** can be performed by checking the balance in a requester's account. To do so, the AWS credentials are set as environment variables:

```
Sys.setenv("AWS_ACCESS_KEY_ID" = "AWSAccessKeyId")
Sys.setenv("AWS_SECRET_ACCESS_KEY" = "AWSSecretAccessKey")

# Test connection to live server
AccountBalance()

# Test connection to sandbox server
AccountBalance(sandbox = TRUE)
```

`AccountBalance()` returns the current balance in U.S. Dollars; for the sandbox, this is always $10,000. The `sandbox` parameter can also be changed globally with `options("MTurkR.sandbox" = TRUE)`.

Data preprocessing with MTurkR

A common workflow for using MTurk involves starting with a messy data structure and wanting some better-structured resulting data structure (within R this is presumably a data frame). To use **MTurkR**, the analyst must break down the messy data structure into a set of individual tasks (HITs), create those HITs via **MTurkR**, allow time for workers to complete assignments, and then collect and review completed assignments before proceeding with the analysis of the resulting data in R. How to achieve this in **MTurkR**? I begin by demonstrating how to create a single HIT and then demonstrating more convenient wrapper functions for creating batches of HITs in bulk.

Creating individual HITs

First, creating a HIT requires registering a HITType, which sets various worker-visible characteristics of the HIT(s), four of which are required and three that are optional:

- Title, short title for the HIT to be displayed to workers (required).
- Description, a description of the HIT to be displayed to workers (required).
- Reward, in U.S. Dollars (required).

[4] **MTurkR** also offers a set of interactive command-line menus for performing **MTurkR** operations without the need to write any code. An add-on package called **MTurkRGUI** (Leeper, 2015) implements an even more robust graphical user interface using the cross-platform **tcltk** package. Additional details about these **MTurkR** features are available in the package documentation and on the **MTurkR** wiki at https://www.github.com/leeper/MTurkR.

[5] Note that MTurk is currently only available to requesters with a United States address and a Social Security number.

[6] As a convenience, by default, all API requests and responses are stored in a tab-separated-value log file in the user's working directory, alongside information about API requests.

- Duration, in seconds (required).
- Keywords, a comma-separated list of keywords used by workers to search for HITs (optional; default is empty).
- Assignment auto-approval delay, a time in seconds which specifies when assignments will automatically be paid if not first rejected (optional; default is 30 days).
- Qualification requirements, a complex structure which controls which workers can complete the HIT (optional; default is none).

To register a HITType, at least the first four characteristics just described need to be defined in a call to `RegisterHITType()`, for example:

```
hittype1 <- RegisterHITType(title = "Tell us something",
                            description = "Answer a single question",
                            reward = "0.05",
                            duration = seconds(days = 1, hours = 8),
                            keywords = "text, answer, question",
                            auto.approval.delay = seconds(days = 1))
```

MTurkR's `seconds()` function provides a convenient way of converting time measurements in days, hours, minutes, or seconds into a total number of seconds. With the HITType created, one can begin creating individual HITs associated with that HITType using `CreateHIT()`.

A HIT consists of a HITType and various HIT-specific attributes, the most import of which is a “question” text specifying the contents of the task as shown to the worker via an HTML iframe on the MTurk worker website. Questions can be specified in one of several ways:

- An HTTPS URL (or “ExternalQuestion”) for a page containing the HIT HTML.
- An “HTMLQuestion” structure, essentially the HTML to display to the worker.
- A “QuestionForm” structure, which is a proprietary markup language used by MTurk.
- A “HITLayoutID” value retrieved from the MTurk requester website[7].

In addition to one of the above question specifications, the other HIT attributes are:

- Duration, the number of assignments to be created for the HIT (required; default 1).
- Expiration, a time specifying when the HIT will expire and thus be unavailable to workers, in seconds (required; no default).
- Annotation, specifying a hidden value that describes the HIT as a reference for the requester (optional; default is empty).

In most cases, specifying an HTMLQuestion is the easiest approach. This simply means writing a complete, HTML5-compliant document creating a web form that will display some material to the worker and allow them to enter and submit answer information to the server. Some examples are installed with **MTurkR**, such as:

```
<!DOCTYPE html>
<html>
 <head>
  <meta http-equiv='Content-Type' content='text/html; charset=UTF-8'/>
  <script type='text/javascript'
   src='https://s3.amazonaws.com/mturk-public/externalHIT_v1.js'></script>
 </head>
 <body>
  <form name='mturk_form' method='post' id='mturk_form'
   action='https://www.mturk.com/mturk/externalSubmit'>
  <input type='hidden' value='' name='assignmentId' id='assignmentId'/>
  <h1>What's up?</h1>
  <p><textarea name='comment' cols='80' rows='3'></textarea></p>
  <p><input type='submit' id='submitButton' value='Submit' /></p></form>
  <script language='Javascript'>turkSetAssignmentID();</script>
 </body>
</html>
```

[7] This is useful for creating HITs using **MTurkR** based on templates created on the MTurk requester website.

Workers will see a rendered version of the HTMLQuestion, specifically a question – "What's up?" – and a multi-line text response they can complete. The JavaScript in the HTMLQuestion is essential for the HIT to behave properly. To setup this HIT in the MTurk system, use `CreateHIT()` passing it the HITTypeId created earlier, making the HIT available for 4 days and setting a private annotation field to remind us about the HIT:

```
f1 <- system.file("templates/htmlquestion1.xml", package = "MTurkR")
hq <- GenerateHTMLQuestion(file = f1)
hit1 <- CreateHIT(hit.type = hittype1$HITTypeId,
                  question = hq$string,
                  expiration = seconds(days = 4),
                  annotation = "my first HIT")
```

At this point, a worker needs to submit the assignment. Once that has happened (this can be checked using `HITStatus()` or `GetHIT(hit = hit$HITId)`), the assignment data can be retrieved through:

```
# Retrieve all assignments for a HIT
a1 <- GetAssignments(hit = hit1$HITId)

# Retrieve all assignments for all HITs for a HITType
a2 <- GetAssignments(hit.type = hittype1$HITTypeId)

# Retrieve a specific assignment
a3 <- GetAssignments(assign = a1$AssignmentId[1])
```

These assignments will be automatically approved after one day (according to the value specified in `auto.approval.delay` when registering the HITType). Assignments can be approved manually using `ApproveAssignment()`:

```
# Approve 1 assignment
ApproveAssignments(assignments = a1$AssignmentId[1],
                   feedback = "Well done!")

# Approve multiple assignments
ApproveAssignments(assignments = a1$AssignmentId)

# Approve all assignments for a HIT
ApproveAllAssignments(hit = hit1$HITId)

# Approve all assignments for all HITs of a HITType
ApproveAllAssignments(hit = hittype1$HITTypeId)

# Approve all assignments based on annotation
ApproveAllAssignments(annotation = "my first HIT")
```

Rejecting HITs works identically to the above but using `RejectAssignments()`. Feedback is optional for assignment approval but required for assignment rejection.[8] Feedback is passed through the `feedback` argument.

Managing crowdworkers with QualificationTypes

One important consideration when creating a HIT is that, by default, every HIT is available to all MTurk workers unless QualificationRequirements have been specified in the `RegisterHITType()` operation. Furthermore, these QualificationRequirements are attached to a HITType, not an individual HIT, so HITs directed at distinct subsets of workers need to be attached to distinct HITTypes.

There are several built-in QualificationTypes that can be used as QualificactionRequirements, including country of residence and various measures of experience on MTurk (e.g., number of HITs completed, approval rate, etc.). To configure a HITType that will only be available to workers in the United States who have completed more than 500 approved HITs, first use `GenerateQualificationRequirement()` to setup a QualificationRequirement structure locally. This involves naming the QualificationTypes to use in the QualificationRequirement, along with "comparators" and "values", which are interpreted as logical statements of the form "Locale is equal to US" and "NumberApproved is greater than 500":

[8] Rejected assignments can also be converted to approved within 30 days of rejection, though the reverse operation is not possible.

```
# Shorthand names of location and approval qualifications
q_names <- c("Locale", "NumberApproved")

# Comparators ("==" for location and ">" for past approvals)
q_comparators <- c("==", ">")

# Qualification values ("US" for location and "500" for past approvals)
q_values <- c("US", 500)

# Convert these values into a QualificationRequirement
qreq2 <- GenerateQualificationRequirement(q_names,
                                          q_comparators,
                                          q_values,
                                          preview = TRUE)
```

This structure is passed as the qual.req argument to RegisterHITType() to create a new HITType with these QualificationRequirements:

```
# Register HITType using the QualificationRequirement
hittype2 <- RegisterHITType(title = "Tell us something",
                            description = "Answer a single question",
                            reward = "0.05",
                            duration = seconds(days = 1, hours = 8),
                            keywords = "text, answer, question",
                            auto.approval.delay = seconds(days = 15),
                            qual.req = qreq2)
```

This attaches a QualificationRequirement to all HITs created within this new HITType, preventing workers who fail to meet the qualifications from working on them (or in this case, given preview = TRUE, even viewing the HITs).[9]

In addition to using the built-in QualificationTypes, workers can also be managed in other ways. One way is to block workers who consistently perform inadequate work using BlockWorkers(). This should be used sparingly, however, as workers who are repeatedly blocked will have their MTurk accounts disabled. A data frame of previously blocked workers is return by GetBlockedWorkers(). UnblockWorkers() is provided to unblock workers. In addition, it is possible to email workers using ContactWorkers() and supply optional bonus payments using GrantBonus(). These can be useful for managing complex projects, incentivizing good work, and inviting well-performing workers to complete new projects.

QualificationRequirements set for a HITType can also be used to manage workers' access to HITs. The built-in QualificationTypes are quite useful for this, but requesters can also create more tailored QualificationTypes based on other criteria. A common use case is to only allow new workers to complete a HIT. The steps to achieve this are: create a new QualificationType, assign different values for that QualificationType to past and new workers, and then create a new HITType using this QualificationType as a QualificationRequirement.

```
# Create the QualificationType
thenewqual <- CreateQualificationType(name = "Prevent Retakes",
                                      description = "Worked for me before",
                                      status = "Active",
                                      auto = TRUE,
                                      auto.value = 100)

# Assign qualification
AssignQualification(qual = thenewqual$QualificationTypeId,
                    workers = hit1$WorkerId,
                    value = "50")

# Generate QualificationRequirement
qreq3 <-
  GenerateQualificationRequirement(thenewqual$QualificationTypeId, "==", "100")
```

[9] HITTypes cannot be edited. If you attempt to create two HITTypes with identical properties, they will be assigned the same HITTypeId. If you modify any attribute, a new HITType will be created. If you have HITs that you would like to assign to a different HITType, use ChangeHITType().

```
# Create HIT, implicitly generating HITType
hit2 <- CreateHIT(question = hq$string,
                  expiration = seconds(days = 4),
                  assignments = 10,
                  title = "Tell us something",
                  description = "Answer a single question",
                  reward = "0.05",
                  duration = seconds(days = 1, hours = 8),
                  keywords = "text, answer, question",
                  auto.approval.delay = seconds(days = 15),
                  qual.req = qreq3,
                  annotation = "my second HIT")
```

To explain what is happening here, a new QualificationType was created that workers can "request" through the MTurk website. If they request it, they will automatically be assigned a score of 100 on the QualificationType. This QualificationType was assigned to all of our workers from the first HIT but at a score lower than the automatically granted value. Next, a QualificationRequirement was created that makes a HIT only available to those with the automatically granted value, and, finally, this was attached to a HITType that is created automatically within the call to `CreateHIT()`. Now 10 new workers can complete this HIT, excluding the worker(s) that completed work on the first HIT.

QualificationTypes and QualificationRequirements on HITTypes allow a requester to manage a large pool of workers in complex ways. Workers that have been assigned scores on a QualificationType can be retrieved using `GetQualifications()`, or modified using `UpdateQualificationScore()`. The attributes of the QualificationType itself can be changed using `UpdateQualificationType()`, and the QualificationType and all associated scores can be deleted using `DisposeQualificationTypes()`.[10] QualificationTypes can also be configured with a "qualification test" that allows workers to submit provisional work as a measure of abilities and then qualifications can be approved/revoked manually based on their responses or even configured with an "AnswerKey" that will automatically evaluate the worker's test performance and assign a score for the QualificationType. Again, the **MTurkR** documentation includes extended examples and possible use cases.

When finished with a HIT and all of its assignment data, it can be deleted from the system using `DisposeHIT()`. This is not a reversible action, so it should be used with caution. HITs will be deleted automatically by Amazon after a period of inactivity, but cleaning up unneeded HITs can be useful given that there is no particularly good way to search for HITs within the system. The `SearchHITs()` operation simply returns a sorted data frame of all HITs.

Creating multiple HITs

In addition to creating single HITs, **MTurkR** offers functionality to manage very large projects involving many HITs. This section describes that functionality in detail.

There are four functions that have been added to **MTurkR** as of v0.6.5 (available on CRAN since 25 May 2015) to facilitate the bulk creation of HITs, for example for the earlier use case of creating a training set of open-ended text responses for a classification algorithm. These functions are wrappers for `CreateHIT()` designed to accept different kinds of input for the `question` argument and cycle through those inputs to create multiple HITs. They are:

- `BulkCreate()` provides a low-level loop around `CreateHIT()` that takes a character vector of question values as input.
- `BulkCreateFromHITLayout()` provides functionality for creating multiple HITs from a HITLayout created on the MTurk Requester website.
- `BulkCreateFromTemplate()` provides higher-level functionality that translates a HIT template and a data frame of input values into a series of HITs.
- `BulkCreateFromURLs()` provides a convenient way of creating multiple HITs from a character vector of URLs.

The last two of these are likely to be the most useful, so extended examples are provided below.

`GenerateHITsFromTemplate()` works from a template HTMLQuestion document containing placeholders for input values and a data frame of values, one set of values per row. An example template is installed with **MTurkR**:

[10] If a QualificationType is requestable but not automatically approved, qualification scores have to be granted manually by the requester. The additional functions `GetQualificationRequests()`, `GrantQualification()`, and `RevokeQualification()` can be used to manage requests.

```
<!DOCTYPE html>
<html>
 <head>
  <meta http-equiv='Content-Type' content='text/html; charset=UTF-8'/>
  <script type='text/javascript'
   src='https://s3.amazonaws.com/mturk-public/externalHIT_v1.js'></script>
 </head>
 <body>
  <form name='mturk_form' method='post' id='mturk_form'
   action='https://www.mturk.com/mturk/externalSubmit'>
  <input type='hidden' value='' name='assignmentId' id='assignmentId'/>
  <h1>${hittitle}</h1>
  <p>${hitvariable}</p>
  <p>What do you think?</p>
  <p><textarea name='comment' cols='80' rows='3'></textarea></p>
  <p><input type='submit' id='submitButton' value='Submit' /></p></form>
  <script language='Javascript'>turkSetAssignmentID();</script>
 </body>
</html>
```

This template contains two placeholders '${hittitle}' and '${hitvariable}'. These placeholders will by replaced by GenerateHITsFromTemplate() with values specified by the hittitle and hitvariable columns in an input data frame, creating a set of unique HITs as one batch.

```
# Create input data frame
inputdf <- data.frame(hittitle = c("HIT title 1", "HIT title 2", "HIT title 3"),
                      hitvariable = c("HIT text 1", "HIT text 2", "HIT text 3"),
                      stringsAsFactors = FALSE)

# Create HITs
bulk1 <-
  BulkCreateFromTemplate(template = system.file("template.html", package = "MTurkR"),
                         input = inputdf,
                         annotation = paste("Bulk From Template", Sys.Date()),
                         title = "Describe a text",
                         description = "Describe this text",
                         reward = ".05",
                         expiration = seconds(days = 4),
                         duration = seconds(minutes = 5),
                         auto.approval.delay = seconds(days = 1),
                         keywords = "categorization, image, moderation, category")
```

The response structure for these functions is a list of single-row data frames. If all HIT creation operations succeed, then the response can easily be converted using do.call("rbind",bulk2) to a data frame, but users will typically only need to examine this structure if errors occurred. Details about the individual HITs can be retrieved at any time using GetHITs() or SearchHITs().

At this point workers need to complete their assignments. Because the same value for the annotation was supplied to all of these HITs, the results for all associated assignments can easily be retrieved using GetAssignments():

```
# Get assignments using annotation
a1 <- GetAssignments(annotation = paste("Bulk From Template", Sys.Date()))
# Get assignments using HITTypeId
a2 <- GetAssignments(hit.type = bulk1[[1]]$HITTypeId)
```

Unfortunately, MTurk does not return the contents of the question parameter with the completed assignments. However HITId is included so it is trivial to merge the input data frame with the assignment data frame allowing the comparison of the original data (e.g., open-ended response text) to the information supplied by workers (e.g., the classification):

```
# Extract HITIds from `bulk1`
inputvalues$HITId <- do.call("rbind", bulk1)$HITId

# Merge `inputvalues` and `assignmentresults`
merge(inputdf, a1, all = TRUE, by = "HITId")
```

BulkCreateFromURLs() behaves similarly but accepts a character vector of URLs to be used as ExternalQuestion values. This function requires a frame.height argument to specify the vertical size of the HIT as shown to workers.[11]

```
bulk2 <-
  BulkCreateFromURLs(url = paste0("https://www.example.com/", 1:3, ".html"),
                     frame.height = 450,
                     annotation = paste("Bulk From URLs", Sys.Date()),
                     title = "Categorize an image",
                     description = "Categorize this image",
                     reward = ".05",
                     expiration = seconds(days = 4),
                     duration = seconds(minutes = 5),
                     auto.approval.delay = seconds(days = 1),
                     keywords = "categorization, image, moderation, category")
```

Addressing problems

Sometimes things go wrong. Perhaps the HITs contained incorrect information or the work being performed is of low quality because of a mistake in the HIT's instructions. When these situations occur, it is easy to address problems using a host of HIT-management functions. To expire a HIT early, simply call ExpireHIT() specifying a HITId, HITTypeId, or annotation value. To delay the expiration of HIT by a specified number of seconds use ExtendHIT() with its add.seconds argument. A call to ExtendHIT() with the add.assignments parameter increases the number of available assignments for the HIT(s).[12]

One other useful set of operations provided by MTurk is a "notification" system that allows requesters to receive messages about various HITType events either via email or to an AWS Simple Queue Service (SQS) queue (see **MTurkR** documentation for examples of the latter). Notifications can be triggered by various events and can be used as an alternative to actively monitoring the status of a HIT vai HITStatus(). Here is an example notification to send an email whenever a HIT of a given HITType expires:

```
n <- GenerateNotification("requester@example.com",
                          event.type = "HITExpired")
SetHITTypeNotification(hit.type = hittype1$HITTypeId,
                       notification = n,
                       active = TRUE)
```

An example of massive-scale photo rating

To demonstrate the ease with which **MTurkR** can be used to preprocess a massive amount of data, I provide an example of a large-scale photo-rating task. Here, I was interested in obtaining a rating of "facial competence" for U.S. politicians compared with ratings of faces from the general U.S. population. Facial competence is said to enhance politicians' electoral success, but previous studies have never compared these to a general population sample. Are politicians generally more facially competent than other individuals? While this is a modest research question, it demonstrates well the immense human effort needed to draw even simple conclusions from messy data structures.

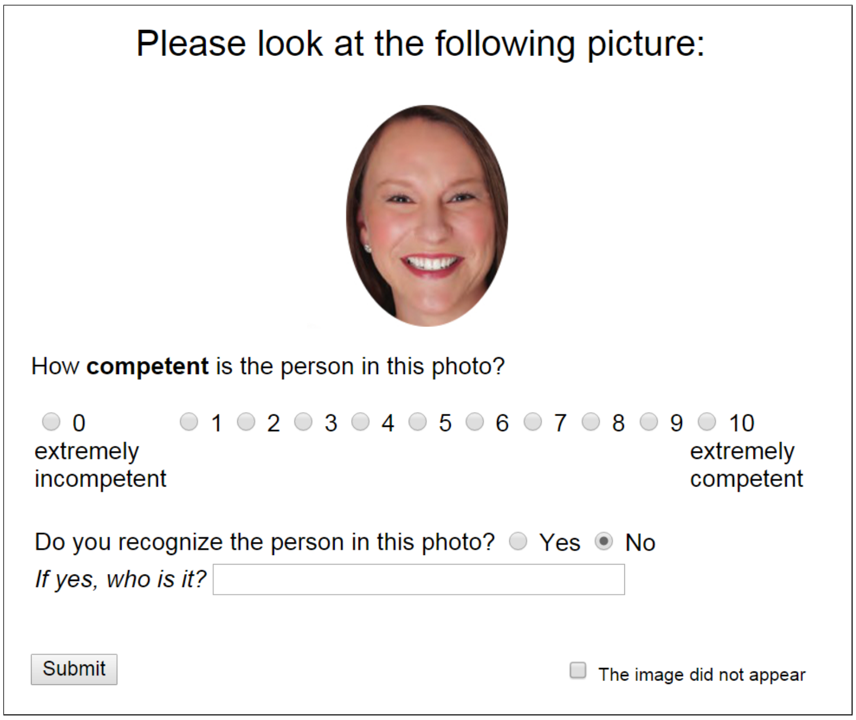
To provide a sampling of politicians' faces, I scraped photos of 533 members of the 113th U.S. Congress from the website of the Government Printing Office. I then combined these photo data with 5000 randomly sampled images from the 10K U.S. Adult Faces Database (Bainbridge et al., 2013), which provides a nationally representative sampling of U.S. faces, and standardized the image size and resolution across all faces.[13] To rate facial competence, I created a simple one-question HIT using HTML (see Figure 1) that displayed one of the faces and asked for a rating of facial competence on a 0 to 10 scale.[14] I include the complete HTML file in the supplemental material for this article.

[11] MTurk displays the page specified by the ExternalQuestion URL inside an HTML iframe on the worker site.

[12] Note that this number must be positive and, therefore, the number of available assignments cannot be reduced. If it is needed to reduce the number of assignments completed for a HIT, the HIT can be expired once the desired number of assignments have been completed.

[13] Complete code to perform the scraping and image processing are provided along with supplemental material for this article at https://github.com/leeper/mturkr-article (http://dx.doi.org/10.5281/zenodo.33595).

[14] The HIT additionally included questions to address possible problems (i.e., a subject recognizes a face or the image did not display properly).



Please look at the following picture:

How **competent** is the person in this photo?

○ 0 extremely incompetent ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 ○ 7 ○ 8 ○ 9 ○ 10 extremely competent

Do you recognize the person in this photo? ○ Yes ● No

If yes, who is it?

□ The image did not appear

Figure 1: Example photo rating HIT.

After uploading all 5533 images to an Amazon Simple Storage Service (S3) bucket, which is a simple cloud storage facility, to make the files publicly available[15] and storing their filenames in a local RDS file, it was trivial to send these images to MTurk workers for categorization. To ensure reliability of the results, each face was rated by 5 workers. Workers were given 45 seconds to rate each face and were paid $0.01 per face. The 27,665 images were rated by a team of 225 U.S.-based workers over a period of 75 minutes. The entire operation cost $412.50. Achieving this required three steps in **MTurkR**: (1) creating a QualificationRequirement to restrict the task to U.S.-based workers with 95% approval ratings, (2) registering a HITType into which the HITs will be created, and (3) the creation of a batch of HITs using `BatchCreateFromURLs()`.

```
# Setup QualificationRequirement
## U.S.-based, 95% approval on HITs
qual <-  GenerateQualificationRequirement(c("Locale", "Approved"),
                                          c("==", ">"),
                                          c("US", 95),
                                          preview = TRUE)

# Register HITType
desc <- "Judge the competence of a person from an image of their face.
  The HIT involves only one question: a rating of the competence of the
  person. You have 45 seconds to complete the HIT. There are several
  thousand HITs available in this batch. If you recognize the person,
  please enter their name in the space provided; your work will still be
  approved even if you recognize the face."
hittype <-
  RegisterHITType(title = "Rate the competence of a person",
                  description = desc,
                  reward = "0.01",
                  duration = seconds(seconds = 45),
                  auto.approval.delay = seconds(days = 1),
                  qual.req = qual,
                  keywords = "categorization, photo, image, rating, fast, easy")

# All faces were loaded into Amazon S3
```

[15] Any public file host could be used, not just S3.

```
s3url <- "https://s3.amazonaws.com/mturkfaces/"
# File names were saved as a character vector locally
faces <- readRDS("faces_all.RDS")
d <- data.frame(face = paste0(s3url,faces),
                stringsAsFactors = FALSE)

# Create 5500 HITs
bulk <- BulkCreateFromTemplate(template = "mturk.html",
                               frame.height = 550,
                               input = d,
                               hit.type = hittype$HITTypeId,
                               expiration = seconds(days = 7),
                               # 5 assignments/face
                               assignments = 5,
                               annotation = "Face Categorization 2015-06-08")
```

Using the specified annotation value, GetAssignments() returns a large data frame with 27670 rows and 25 columns:

```
a <- GetAssignments(annotation = "Face Categorization 2015-06-08")
dim(a)
# [1] 27670    25
names(a)
#  [1] "AssignmentId"         "WorkerId"               "HITId"
#  [4] "AssignmentStatus"     "AutoApprovalTime"       "AcceptTime"
#  [7] "SubmitTime"           "ApprovalTime"           "RejectionTime"
# [10] "RequesterFeedback"    "ApprovalRejectionTime"  "SecondsOnHIT"
# [13] "competent"            "recognized"             "name"
# [16] "face"                 "condition"              "browser"
# [19] "engine"               "platform"               "language"
# [22] "width"                "height"                 "resolution"
# [25] "problem"
```

Most of the columns contain metadata for identifying each assignment (AssignmentId, WorkerId, HITId), metadata about the completion of the assignment (AssignmentStatus, AutoAprpovalTime, AcceptTime, SubmitTime, ApprovalTime, RejectionTime, RequesterFeedback, ApprovalRejectionTime, SecondsOnHIT), and then several columns displaying responses to the three HIT questions displayed to the workers: competent, recognized, and name. The names of these variables are given by the name attribute of the radio buttons used in the HTMLQuestion form. The data frame also contains additional variables that record metadata about the worker's browser, which were recorded automatically via Javascript.

As noted earlier, a limitation of the MTurk API is that it does not return information about the values of variables replaced in the templating process, so it can be difficult to identify which assignment(s) correspond to which input values. To circumvent this limitation, this HIT template was designed to use the ${face} variable twice: once to actually display the image to the worker and once to record its value in a hidden field called face in the HTMLQuestion form. As a result, this variable becomes available to us in the results data frame.

Setup in this way, it becomes trivial to analyze facial competence ratings of politicians and those from the general population sample. To perform the analysis, I simply conducted a Mann-Whitney-Wilcoxon test for a difference in competence ratings between the faces of politicians and non-politicians. (In these data, politicians' photos were identified by a simple pattern matching file name. This would have more easily been done with a hidden HTML variable when creating the batch.) So, I extract the two variables from the assignment data frame, convert them to numeric, and perform the test:

```
competence <- as.numeric(a$competent)
politician <- as.numeric(grepl("[[:digit:]]{2}-[[:digit:]]{3}", a$face))

round(prop.table(table(politician, competence), 1), 2)
#
#           competence
# politician    0    1    2    3    4    5    6    7    8    9   10
#          0 0.03 0.03 0.04 0.07 0.11 0.13 0.17 0.17 0.16 0.06 0.02
#          1 0.01 0.01 0.02 0.04 0.07 0.12 0.19 0.20 0.22 0.09 0.04

wilcox.test(competence ~ politician)
```

```
#
#         Wilcoxon rank sum test with continuity correction
#
# data:  competence by politician
# W = 26886000, p-value < 2.2e-16
# alternative hypothesis: true location shift is not equal to 0
```

Politicians do appear to have higher facial competence. While this is a fairly trivial analytic conclusion, it demonstrates the ease with which crowdsourced human intelligence can be leveraged to preprocess a massive amount of data, translating messy sources into easily analyzed data. Because crowdsourcing is inherently massively parallel, it dramatically reduces the amount of time needed to parse a rough data source. In this case, the MTurk workers created the completed dataset in about 75 minutes. Were a single individual to attempt this task alone and it took (as a generous estimate) only 5 seconds to categorize each face, the task would be completed in 38.4 hours, or about 31-times as long as with MTurk.

Conclusion

This paper has described the MTurk platform and offered an introduction to the R package **MTurkR** focused on preprocessing of messy data for immediate use in R. In short, **MTurkR** provides a stable, well-developed R interface to one of the largest crowdsourcing sites presently available. The package has been developed and refined for more than three years, has extensive in-package and online documentation, and is incredibly easy to use. By providing a low-level wrapper to the Amazon Mechanical Turk API, it also means that **MTurkR** could serve well as the basis for much more sophisticated R applications that leverage human intelligence as an enhancement to the computational features already available in R.

Bibliography

Amazon.com. Amazon Mechanical Turk getting started guide, 2012. URL http://docs.amazonwebservices.com/AWSMechTurk/latest/AWSMechanicalTurkGettingStartedGuide/Welcome.html?r=4925. [p276]

W. A. Bainbridge, P. Isola, and A. Oliva. The instrinsic memorability of face photographs. *Journal of Experimental Psychology: General*, 142(4):1323–1334, 2013. doi: 10.1037/a0033872. [p284]

A. J. Berinsky, G. A. Huber, and G. S. Lenz. Using Mechanical Turk as a subject recruitment tool for experimental research. Unpublished paper, 2010. [p277]

M. Buhrmester, T. Kwang, and S. D. Gosling. Amazon's Mechanical Turk: A new source of inexpensive, yet high-quality, data? *Perspectives on Psychological Science*, 6(1):3–5, Feb. 2011. doi: 10.1177/1745691610393980. [p277]

J. J. Chen, N. J. Menezes, and A. D. Bradley. Opportunities for crowdsourcing research on Amazon Mechanical Turk. Unpublished paper, 2011. [p276]

A. Kittur, E. H. Chi, and B. Suh. Crowdsourcing user studies with Mechanical Turk. In *CHI 2008 – Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, page 453, New York, New York, USA, 2008. ACM Press. doi: 10.1145/1357054.1357127. [p277]

T. J. Leeper. Crowdsourcing with R and the MTurk API. *The Political Methodologist*, 20(2):2–7, 2013. [p277]

T. J. Leeper. *MTurkRGUI: A Graphical User Interface for MTurkR*, 2015. URL https://CRAN.R-project.org/package=MTurkRGUI. R package version 0.1.5. [p278]

T. J. Leeper. *MTurkR: R Client for the MTurk Requester API*, 2016. URL https://www.github.com/leeper/MTurkR. R package version 0.7.0. [p276, 278]

W. Mason and S. Suri. Conducting behavioral research on Amazon's Mechanical Turk. *Behavior Research Methods*, 44(1):1–23, Mar. 2012. doi: 10.3758/s13428-011-0124-6. [p277]

J. Ooms. *curl: A Modern and Flexible Web Client for R*, 2016. URL https://CRAN.R-project.org/package=curl. R package version 0.9.6. [p278]

G. Paolacci, J. Chandler, and L. N. Stern. Running experiments on Amazon Mechanical Turk. *Judgment and Decision Making*, 5(5):411–419, 2010. [p277]

L. A. Schmidt. Crowdsourcing for human subjects research. In *CrowdConf 2010*, San Francisco, CA, 2010. [p276]

D. Temple Lang. *XML: Tools for Parsing and Generating XML within R and S-Plus*, 2012. URL `http://CRAN.R-project.org/package=XML`. R package version 3.9-4.1. [p278]

Thomas J. Leeper
Department of Government
London School of Economics and Political Science
London, United Kingdom
`thosjleeper@gmail.com`