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, taking advantage of a highly developed package for R. While MTurk has many uses, especially to users from the social sciences, this article focuses on ways that MTurk can help users preprocess "messy" data. The package, **MTurkR**, provides users access to the full functionality of the MTurk platform, reducing barriers to entry for researchers to begin research projects through the platform, while also providing human intelligence to supplement the computational capacity of R. This article outlines MTurk and its utility for crowdsourcing, describes the MTurkR package, and demonstrates how to gather and manage data from MTurk using some of the package's core functionality. As an example of a massive data preprocessing, an image rating task involving 225 workers and more than 5500 images is demonstrated using just three function calls.

People use R because it is extensible, robust, and free. But R is still just a computing environment, with the computational limitations inherent in any such environment. It can do many things, but doing those many things generally requires data structures that are R-ready or that can be parsed with one of R's many input functions. Yet R users working in many real-world situations are sometimes faced with data that are not R-ready: handwritten survey responses, digitized texts that cannot be read by optical character recognition, images, etc. Or, alternatively, a data analyst may face machine readable data but require human interpretation to categorize, translate, or code those data. For example, someone wishing to build an automated classifier may require a human-categorized training set that is not yet available.

In such cases, considerable effort needs to be put into translating those data into structures or values that can be processed computationally. Making the leap from these raw data to R data structures can therefore entail considerable human labor, usually on the part of the analyst. Such needs for human labor in data preprocessing has provoked interest in online crowdsourcing platforms (Schmidt, 2010; Chen et al., 2011) to provide human intelligence to tasks that cannot be easily accomplished through computation alone.

This paper describes the use of the MTurkR package to leverage the Amazon Mechanical Turk (MTurk) crowdsourcing platform to bring human intelligence into R. MTurk has the potential to reduce R users' human labor costs as well as time spent cleaning and preprocessing data, and the MTurkR package makes the start-up costs for prospective MTurk users relatively minimal. The article begins by laying out the need for occasional human intelligence in data preprocessing, then describes MTurk and its vocabulary, and introduces the MTurkR package (Leeper, 2012). The remainder of the article describes the core functionality of the MTurkR package, showing how it can be used in practice to accomplish — among other things — data preprocessing.

The Need for Human Intelligence

Some tasks cannot be automated. Others can be automated only with difficulty. In these situations, it can be helpful to process certain data-related tasks manually, but doing so comes at the cost of time, money, and effort. R currently only provides means of computation but has no means of leveraging human intelligence (other than the user's). Archetypal needs for human intelligence include the collection of data which cannot be automated (e.g., online data that lack the well-defined structure to be scraped and parsed as HTML, XML, JSON, etc.), the preprocessing of "messy" data in formats that cannot be directly analyzed (e.g., 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., long, 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).

Because the translation of these types of "messy" data into R data structures typically requires the time and effort of the R user, resources which do not scale well as the size of datasets increase, facilities readily available through R to recruit and utilize human intelligence is therefore a significant resource with a large number of plausible use cases. Crowdsourcing data preprocessing needs is an obvious way to obtain this human intelligence.[1] Amazon Mechanical Turk (MTurk) stands out as one of the largest and most useful crowdsourcing platforms currently available, and its use is facilitated by a

[1]Crowdsourcing being an etymological play on "outsourcing" involving the distribution and partition of tasks to crowds rather than individual contractors.

powerful API that is now accessible through the MTurkR package. By crowdsourcing data processing, R users can reduce the time and effort they spend preprocessing data, while also leveraging multiple sources of human intelligence to improve the reliability and speed of such efforts.

MTurk: Introduction and Core Concepts

Amazon Mechanical Turk is a crowdsourcing platform designed by Amazon.com 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). Regardless of the intentions behind the creation of MTurk, the service provides R users with a useful infrastructure for data preparation, data cleaning, and the human translation of real-world data into R data structures. Because MTurk provides the web application for recruiting, paying, and managing human workers, the effort necessary to move one's own work or that which would be done by paid assistants into the cloud is relatively effortless and 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, 2011; Kittur et al., 2008), more recent attention has also emerged among social scientists who see MTurk's pool of workers as more than an affordable source of human labor (Buhrmester et al., 2011; Berinsky et al., 2010; Paolacci et al., 2010). This article provides a sufficiently general overview of MTurk and the MTurkR package to enable its use for a variety of purposes, but focuses primarily on the uses of MTurk for data preprocessing.[2]

Key Terms and Concepts

MTurk connects *requesters*, who are willing to pay *workers* to perform a given task or set of tasks at a given 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 the one 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 a 5000 text statements (e.g., free response answers on a survey) 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 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 operate on basic market principles of supply and demand. They can 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.[3] A requester can offer as low as \$0.005 per assignment, but if other requesters offer HITs that pay more per assignment, workers can choose to take their labor elsewhere. Similarly, requesters can pay any higher amount they want per assignment, but that may not be cost-effective given the market forces operating within the MTurk marketplace. Workers increasingly demand competitive wages, at a rate of at least U.S. minimum wage.

Once a worker completes a HIT, the requester can *review* the assignment — that is, they can see the responses or answers provided by the worker to whatever prompts the requester set up in the HIT, which the requester can either *approve* or *reject*. If the work is deemed satisfactory, it should be approved and the requester then pays the worker the predetermined per-assignment price for the HIT (and no more or no less; the price is fixed in advance). 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. If work is unsatisfactory, the requester should reject work and thereby deny payment (but has to justify that rejection to the worker), freeing the completed assignment for completion by another worker. MTurk also charges a surcharge on top of all worker payments. This review process can also be avoided entirely by granting rewards automatically or by implementing an automated ReviewPolicy process (discussed later on).

[2]Users specifically interested in social science survey and experimental applications should consult Leeper (2013).

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

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 skill).

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.[4]

MTurk API and Other Packages

Amazon provides a set of Software Development Kits (SDKs) written for Python, Perl, Ruby, etc. that make use of the Requester API. Unfortunately, no officialy supported client exists for R. The **MTurkR** package therefore provides ready access to the API for those who want to crowdsource from within a familiar R environment. A major advantage of MTurkR is that it can control the complete MTurk workflow, from submitting "messy" data to MTurk, reviewing work completed by workers, and retrieving completed work as an R data.frame. MTurkR also includes both an interactive command-line interface and a sophisticated graphical user interface written in Tcl/tk, providing incredibly simple crowdsourcing capabilities with almost no R knowledge required.

The MTurkR Package

Before using MTurk or MTurkR, one needs to have an MTurk requester account, which can be created at `http://www.mturk.com`. It is also helpful from a practical perspective to have a worker account (e.g., to confirm one's own HITs look and feel as intended), so it makes sense to register both right away. In order to use MTurk as a requester, you need to deposit money into your requester account. To use MTurk as a worker, you need to a link a bank account to the system so that you can be paid for the work performed. It is helpful to perform some work simply to familiarize yourself with the system and how workers will view the information you create as a requester. Performing an assignment for yourself as a requester also allows you to test some of the API features provided by MTurkR.

Before using MTurkR, one needs to retrieve Amazon Access Keys from `https://console.aws.amazon.com/iam/home?#security_credential`. The *keypair* is a linked *Access Key ID* and a *Secret Access Key*. In combination, these security credentials allow MTurkR to access the API. It is possible to create multiple keypairs per requester account (e.g., to share a requester account) and to replace old keypairs with new ones for security reasons. In MTurkR, the keypair is a two-element character vector with the Access Key ID as the first element and the Secret Access Key as the second element. This keypair is used to authenticate API requests.

The MTurk API is somewhat dated by comparison to contemporary web development standards, relying exclusively on HTTP POST requests and supplying response structures only in XML. MTurkR converts all responses into R data structures (typically data.frames) that can be directly used in analysis with no need for manual conversions to R-readable data.[5] In the event an API request fails, error reporting information is returned instead of the standard data structure. Aside from problems with authentication or attempting to retrieve HIT data that does not exist, users are very unlikely to encounter API-related errors because MTurkR was programmed to capture almost all potential request errors within R prior to attempting an API request.

Using MTurkR

The simplest MTurkR request is to check the balance in one's account. To do so, first define the two-element keypair vector with a call to `options(MTurkR.keypair = c("AWSAccessKeyId","AWSSecretAccessKey"))`. These values can also be passed as a `keypair` argument to any MTurkR function, but they are otherwise automatically retrieved from the value stored in `options()`. A call to `AccountBalance()`,

[4]As an example, identical HITs created in the sandbox and live server will have different HITIds that cannot be used on the opposite servers.

[5]HTTP requests are implemented using **curl** (Ooms et al., 2015) and XML parsing is provided by **XML** (Temple Lang, 2012). The raw XML responses are stored, by default, in a tab-separated value log file in the user's working directory, alongside information about API requests.

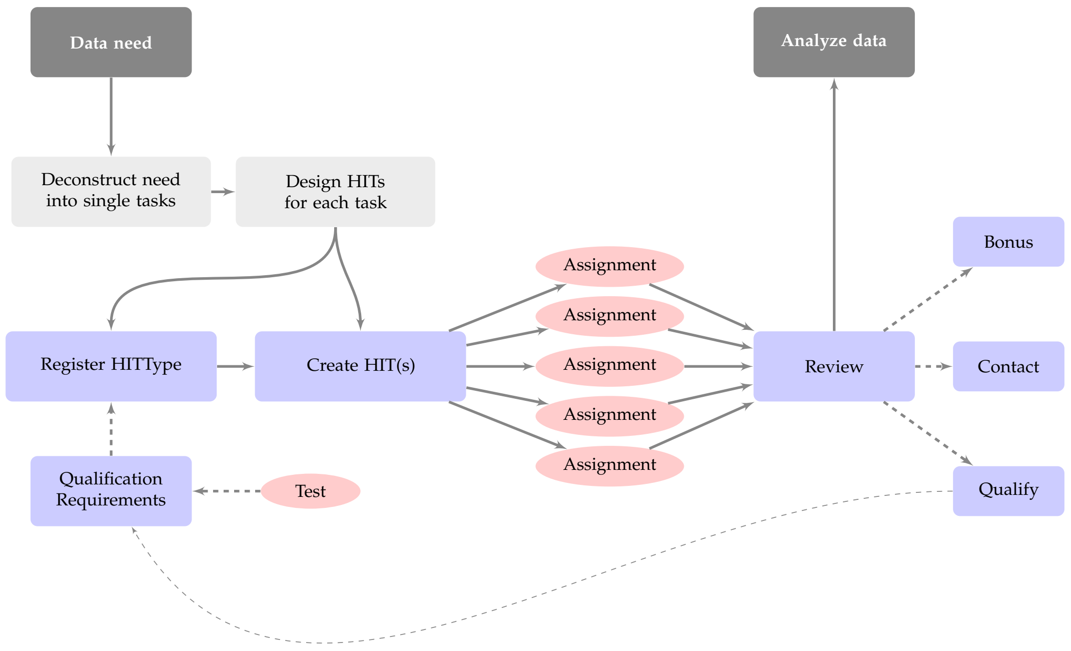


Figure 1: A Typical MTurk Workflow

with no arguments, returns a "Hello world!"-style response informing you of the currency balance in your account in U.S. Dollars.[6] All MTurkR functions by default send requests to the live server, but can be sent to the sandbox by adding a `sandbox=TRUE` parameter to the function call (e.g., `GetAccountBalance(sandbox=TRUE)` always returns a balance of $10,000.00). The sandbox parameter can also be changed globally with `options("MTurkR.sandbox" = TRUE)`. For most functions, the object returned by an MTurkR function is a data.frame containing one or more records.[7]

Data Preprocessing with MTurkR

A common workflow for using MTurk involves a data need and some desired data structure (presumably a data.frame) (see 1). Any use of MTurk starts with a data need, which the researcher must break down into a set of individual tasks (i.e., HITs), create those HITs via MTurkR, allow time for workers to complete assignments, and then collect and review completed assignments before proceeding with analysis of the resulting data in R.

No matter the form of the input data need, MTurkR will always produce a data.frame of results, with one worker's assignment data per row. Imagine, for example, a use cases involving 1000 open-ended text responses to a survey that need to be classified into one of ten categories. If each piece of text constitutes a HIT and perhaps three assignments are made available for each HIT (i.e., three workers per text), then the resulting data returned by MTurkR will be 3000 row data.frame, with columns indicating a vector of 3000 unique AssignmentId for each row and a vector of 1000 HITId's repeated three times.

How do we achieve this in MTurkR? I begin by demonstrating how to create a single HIT and then demonstrate more convenient wrapper functions for creating batches of HITs in bulk.

[6]The API does not allow you to add funds to your account, which must therefore be done through the web interface: `https://requester.mturk.com/mturk/prepurchase`.

[7]Details for specific functions can be found in the MTurk documentation (Leeper, 2012).

Creating Individual HITs

Creating a HIT requires first registering a HITType, which sets various characteristics of one or more HITs and groups all HITs of that type visually in the MTurk worker site. One must supply four required and up to three optional characteristics:

- 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)
- Duration, in seconds (required)
- Keywords, a comma-separated list of keywords used by workers to search for HITs
- Assignment Auto-Approval Delay, a time in seconds which specifies when assignments will automatically be paid if not first rejected
- Qualification Requirements, a complex structure which controls which workers can complete the HIT

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))
```

The `seconds()` function provides a convenient way of converting days, hours, minutes, and seconds into a total number of seconds. With the HITType created, one can begin creating individual HITs associated with that HITType using `CreateHIT()`.[8]

A HIT consists of a HITType and various HIT-specific attributes:

- A "question" text (required), consisting of one of the contents of the task to be displayed to the worker in an HTML iframe on the MTurk worker website:
  - 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
- 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

In most cases, specifying an HTMLQuestion is the easiest approach. This simply means writing a complete, HTML5-compliant document including a web form that will display some material to the worker and allow them to enter answer information and submit it to the server. A basic example is installed with MTurkR:

```
<!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'/>
```

[8] All of the arguments to `RegisterHITType()` can also be used atomically within `CreateHIT()` instead as a convenience.

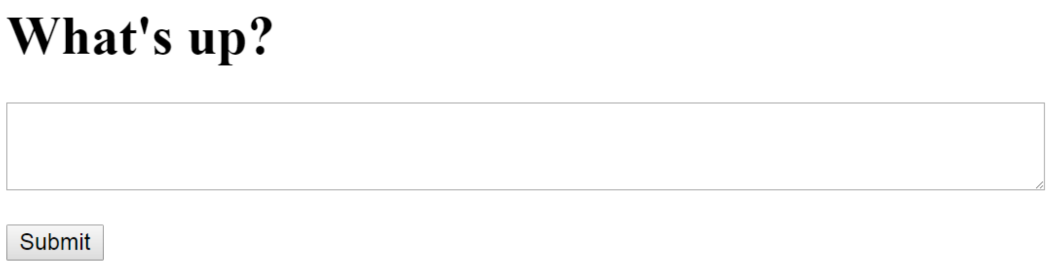


Figure 2: A Very Basic HIT

```
  <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>
```

To create this HIT in the MTurk system, we use `CreateHIT()`:

```
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")
```

This HIT is made available for four days, with one assignment (the default), and we have added a private annotation field reminding us that this is our first HIT. Workers will see a rendered version of the HTMLQuestion, specifically a question — "What's up?" — and a multi-line text response they can complete (see 2). The (Javascript in the HTMLQuestion is essential for the HIT to behave properly.)

At this point, we need to wait to allow a worker to submit the assignment. Once that has happened (and we can check using `HITStatus()` or `GetHIT(hit = hit$HITId)`), then we can retrieve assignment data:

```
# 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 we specified in `auto.approval.delay` when registering the HITType). We can also approve the assignments 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. Rejected assignments can also be converted to approved within 30 days of the rejection, though the reverse operation is not possible.

One important consideration when creating a HIT is that every HIT is, by default, available to all MTurk workers unless Qualification Requirements have been specified in the `RegisterHITType()` operation. Furthermore, these Qualification Requirements 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 500 HITs, we can do the following:

```
qreq2 <- GenerateQualificationRequirement(c("Locale","NumberApproved"),
                                          c("==",">"),
                                          c("US",500),
                                          preview = TRUE)
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 (or in this case, given `preview = TRUE`, even viewing the HITs).[9]

In addition to using the built-in QualificationTypes, you can also manage workers 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. You can see a data.frame of previously blocked workers using `GetBlockedWorkers()` and unblock workers using `UnblockWorkers()`.

In addition, it is possible to send an email to workers using `ContactWorkers()` and supply additional bonus payments on top of the automatic HIT reward using `GrantBonus()`. These can be useful for managing complex project, incentivizing good work, and inviting well-performing workers to complete new projects.

To simply manage workers' access to HITs, QualificationRequirements instead. The built-in QualificationTypes are quite useful for this, but you may also want to create more tailored QualificationTypes based on workers' performance only on your HITs.[10]

To monitor workers, you can use `GetWorkerStatistic()` to retrieve performance information for each worker on your HITs. A number of statistics are available and can be conveniently retrieved in one go as follows:

```
# retrieve a worker report
WorkerReport("AnExampleWorkerId")

# retrieve a specific statistic instead
GetWorkerStatistic("AnExampleWorkerId", "NumberAssignmentsApproved")
```

Each of these statistics (and the worker report) can be retrieved for one of several periods: "LifeToDate," "ThirtyDays," "SevenDays," and "OneDay" using a call including the `period` parameter, e.g. `GetWorkerStatistic(workerid,period="SevenDays")`. The default time period is to return the statistic for "LifeToDate."

These statistics and other information can be used to create customer, requester-specific QualificationTypes that will further restrict which workers can complete HITs. A common use case is

[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()`.

[10]The built-in QualificationTypes, such as number of HITs approved, reflect workers' performance on all requesters' HITs.

to only allow new workers to complete a HIT. To achieve this, we need to create a new QualificationType, assign that QualificationType to past 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")

# 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, we create a new QualificationType that workers can "request" through the MTurk website. If they request it, they will automatically be assigned a score of 100 on the QualificationType. We then assign this QualificationType to all of our workers from our first HIT but at a score lower than the automatically granted value. We next create a QualifciationRequirement that will make a HIT only available to those with the automatically granted value, and we finally attach this to a HITType that we create atomically within our call to `CreateHIT()`. Now 10 new workers can complete this HIT, excluding the worker(s) that completed work on our first HIT.

This basic infrastructure of QualificationTypes and QualificationRequirements allows a requester to manage a large pool of workers in complex ways, making a HITType avaiable to very specific types of workers as needed. Workers scores on a QualificationType can be retrieved using `GetQualifications()`, or modified using `UpdateQualificationScore()`. The attributes of the QualificationType itself can changed using `UpdateQualificationType()`, and the QualificationType and all associated scores can be deleted using `DisposeQualificationTypes()`. If a QualificationType is requestable but not automatically approved, qualification scores have be granted manually by the requester using additional functions `GetQualificationRequests()`, `GrantQualification()`, and `RevokeQualification()` can be used to manage requests. Alternatively, QualificationTypes can 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 using the preceding functions, or the QualificationType can also be setup 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 we are done with HIT and all of its assignment data, we can delete it 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 an ordered list of all HITs.

Creating Multiple HITs

With the basics of HIT creation and worker management introduced, it is now possible to discuss how to manage very large projects involving many HITs. To create multiple related HITs, one could simply loop over `CreateHIT()` many times and then use the HITId, HITTypeId, or annotation values for those

HITs to retrieve work and approve assignments. As of MTurk v0.6.5, however, several convenience functions have been added to make it even easier to create many related HITs. This section describes that functionality in detail.

As a brief aside, it is important to note that approving work can become very time consuming as project size increases. Because of the way the MTurk requester API is designed, approving or rejection an assignment can only be done one-at-a-time. As such, it is very useful to configure short automatic approval times for large projects, so that work is simply approved automatically without the need to manually evaluate assignments.

There are four functions that have been added to MTurkR to facilitate the bulk creation of HITs, for example for the earlier use case of creating a supervised classification algorithm from a training set of open-ended text responses. 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 most useful, so I provide extended examples below. Note that all four functions require an `annotation` argument in order to be able to easily identify them in the system. This can be a vector of unique values.

`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:

```
<!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}`'. `GenerateHITsFromTemplate()` will replace these placeholders with values specified by the '`hittitle`' and '`hitvariable`' columns in an input data.frame, creating set of unique HITs as one batch.

```
# load template
temp <- system.file("template.html", package = "MTurkR")

# 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 = temp,
                       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. The reason for this is backwards compatibility with earlier behavior of `CreateHIT()`. If all HIT creation operations succeed, then the response can easily be converted to a data.frame using `do.call("rbind",bulk2)`.

The results from a set of multiple HITs 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:

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

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

This provides a single data.frame that can contains both the original data (e.g., open-ended response text) and information supplied by the workers as columns.

`BulkCreateFromURLs()` works similarly but instead accepts a character vector of URLs to be used as ExternalQuestion values. This function requires a `frame.height` argument to specify the size of the iframe in which the HIT pages will be shown.

```
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. It returns a data.frame confirming whether each of the requests to expire a particular HIT was valid.

To delay the expiration of HIT, `ExtendHIT()` with a `add.seconds` parameter extends the specified HIT(s) by the specified number of seconds.

A call to `ExtendHIT()` with the `add.assignments` parameter specified increases the number of available assignments for the specified HIT by the specified increment. Note that this number must be positive and, therefore, the number of available assignments cannot be reduced. If you need to reduce the number of assignments completed for a HIT, simply expire the HIT once the desired number of assignments have been completed.

A (likely less common) situation is where a requester needs to change the HITType of a given HIT. Recall that the title, description, reward amount, and Qualification Requirements (if applicable) of a given HIT are determined by the HITType, not the HIT per se. So, if you desire to change these characteristics of a currently assignable HIT, you would need to change the HITType of the HIT, using `ChangeHITType()`. This, too, can be called on a single HITId or a group of HITs specified by HITTypeId or annotation. This situation might emerge when a HIT is not being completed quickly enough, so a requester lowers or removes Qualification Requirements, changes the description, and/or increases the worker reward.

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 Queue. Notifications can be triggered by various events: 'AssignmentAccepted,' 'AssignmentAbandoned,' 'AssignmentReturned,' 'AssignmentSubmitted,' 'HITReviewable,' or 'HIT-Expired.' These notifications might can be used as an alternative to actively monitoring the status and completion of a HIT (such as through `HITStatus()`).

Notifications are only sent if configured for a given HITType — that is, notifications are never sent by default *and* can only be configured to be sent for all HITs of a given HITType. Configuring a notification requires two steps: (1) generating a notification data structure, and (2) activating the notification (though the first step can be specified atomically within the second step). An example notification involves an email sent to the request whenever a HIT expires:

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

When this HIT expires, an email containing the HITId will be sent to the specified email address. The process of sending notifications to an SQS queue is more complicated, but details are provided in the MTurkR documentation.

An Example of Massive-Scale Photo Rating

To demonstrate the ease with which MTurk workers can be used to preprocess a massive amount of data, I provide a simple example of a massive-scale photo rating task. Here, I was interested in obtaining a rating of "facial competence" for U.S. politicians compared with the ratings of faces from the general U.S. population as a whole. Facial competence is said to enhance politicians' electoral success, but previous studies have always relied on samples of politicians only and never compared these to a general population sample. Are politicians generally more facially competent than other individuals?

To provide a sampling of politicians' faces, I scraped photos of 533 members of 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) and standardized the image size and resolution across all faces.[11]

To rate facial competence, I created a simple one-question HIT using HTML (see Figure 3) that displayed one of the faces and asked for a rating of facial competence on a 0 to 10 scale.[12] I include the complete HTML file in supplemental materials for this article.

After uploading all 5533 images to an S3 directory 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 a team of 225 U.S.-based workers over a period of 75 minutes. The entire operate cost $412.50.[13]

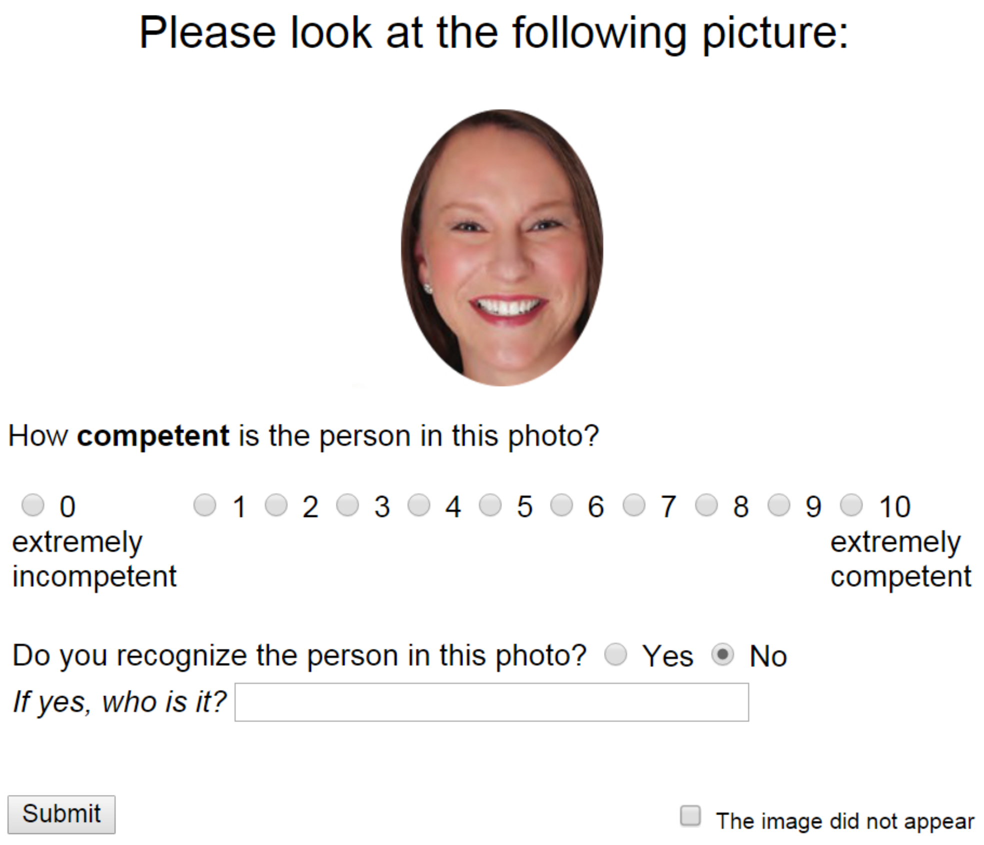
The entire code necessary to implement this project is shown below. It involves three steps: (1) creating a QualificationRequirement to restrict the task to U.S.-based workers with 95% approval ratings, (2) registering a HITType into which the HITs will be created, and (3) the creation of a batch of HITs using `BatchCreateFromURLs()`.

```
library("MTurkR")
```

[11]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).

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

[13]Note that AWS has since increased its pricing slightly, so the code shown here would now cost more than this.



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?

Figure 3: Example Photo Rating HIT

```
# Setup Qualification Requirement
## 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
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")
```

The bulk object contains a list of details about each HIT. Once the crowdsourcing is completed, it is trivial to retrieve the results using `GetAssignments()` based on the `annotation` argument. The result is a large data.frame with 27670 rows and 25 variable 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 each identifying each unique assignment (AssignmentId, WorkerId, HITId), metadata about the completion of the assignment (AssignmentStatus, AutoApprovalTime, 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 specifying by the name atttribute of the radio buttons used in the HTMLQuestion form. Note that this HIT also includes a number of additional variables that record metadata about the worker's browser, which were recorded automatically via javascript.

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 makes it difficult to identify which assignment(s) correspond to which input values. One way to get around this is to merge the input values (e.g., the d data.frame in the above code) with the HITIds from the bulk list and finally combine these with the a assignment data.frame. An easier strategy adopted here was to simply use the ${face} variable twice in the template: once to actually display the image and one 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 assignment 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, we simply conduct a *t*-test for the mean difference in competence ratings between politicians' and non-politicians' faces. 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, we 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))
t.test(competence ~ politician)
#
#         Welch Two Sample t-test
#
# data:  competence by politician
# t = -18.748, df = 3473.2, p-value < 2.2e-16
# alternative hypothesis: true difference in means is not equal to 0
# 95 percent confidence interval:
#  -0.8456886 -0.6855561
# sample estimates:
# mean in group 0 mean in group 1
#        5.741885        6.507508
```

While this is a fairly trivial analytic exercise, 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. The crowdsourced method, while perhaps more expensive, delivers data in a fraction of the time thereby allowing the analyst to focus on statistics rather than data wrangling.

Conclusion

This paper has described the MTurk platform and offered an introduction to MTurkR focused on preprocessing of messy data for immediate use in R. While this article has laid out some of the core functionality of MTurkR, the package provides many additional features not described here. Among the most useful to data scientists may be the ability to develop MTurk-based applications that rely on "Review Policies" that automatically evaluate the work submitted by MTurk workers. These Review Policies can approve and reject work or even extend the number of assignments for a HIT based on known correct answers included in a HIT, plurality agreement among workers, or both. Such applications could easily facilitate the preprocessing of complex or controversial data without the need for substantial human intervention to oversee the preprocessing stage.

Another important set of MTurkR functionality not described in this article is Tcl/tk-based graphical "wizard" interface provided as part of the package. The wizard helps users, especially MTurk novices or those who do not want to use the standard R command line interface, to perform an array of operations on MTurk through a series of interactive, point-and-click menus. Functionality of the wizard remains under development, but currently supports nearly all of the functionality of the MTurkR package. Interested users should consult the MTurkR documentation for more information about the wizard. Additional details about MTurkR are available in the package documentation and on the MTurkR wiki: https://www.github.com/leeper/MTurkR.

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`. [p]

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. [p]

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

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. ISSN 1745-6916. doi: 10.1177/1745691610393980. URL `http://pps.sagepub.com/lookup/doi/10.1177/1745691610393980`. [p]

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

A. Kittur, E. H. Chi, and B. Suh. Crowdsourcing user studies with Mechanical Turk. In *CHI 2008*, page 453, New York, New York, USA, 2008. ACM Press. ISBN 9781605580111. doi: 10.1145/1357054.1357127. URL `http://portal.acm.org/citation.cfm?doid=1357054.1357127`. [p]

T. J. Leeper. MTurkR: Access to Amazon Mechanical Turk requester API, 2012. URL `https://www.github.com/leeper/MTurkR`. [p]

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

W. Mason and S. Suri. Conducting behavioral research on Amazon's Mechanical Turk. Unpublished paper, June 2011. URL `http://www.ncbi.nlm.nih.gov/pubmed/21717266`. [p]

J. Ooms, H. Wickham, and RStudio. *curl: A Modern and Flexible Web Client for R*, 2015. [p]

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

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

D. Temple Lang. *XML: Tools for parsing and generating XML within R and S-Plus*, 2012. URL `http://cran.r-project.org/package=XML`. [p]

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