Guide for Running Amazon MTurk Studies Using https://github.com/alicexue/MTurk

Created by Alice Xue, 11/29/2017
Last Modified: 6/25/2018

This guide provides explanations and instructions for adding studies to the repository. See the documentation in the code for more detailed information about the scripts and their methods.

- **Overview:**
  - Backend: data is stored in csv files using Python Pandas data frames
  - Middleware: Python, Flask framework
  - Frontend: HTML, Jinja templates, JavaScript (the tasks)
- **References:**
  - JavaScript: JavaScript documentation
  - Flask: Flask blueprints and structure
  - MTurk: Concepts, Developer Guide, AWS MTurk Commands, FAQ, HIT Lifecycle, Code samples for creating external HIT
- **Overview of repository structure – folders (*) and files**
  - app.py
  - store_data.py
  - manage_subject_info.py
  - static *
    - stim *
    - components.js
    - run_auction.js
    - run_choicetask.js
  - templates *
    - <project_name> *
      - <task>.html
      - <task>_instructions.html
      - consent_form.html
    - 404.html
    - 500.html
    - accept_hit.html
    - check_eligibility.html
    - dummy_hit.html
    - feedback.html
    - repeat_error.html
    - return_dummy_hit.html
    - return_hit.html
    - thankyou.html
    - unauthorized_error.html
  - <project_name> *
    - __init__.py
    - project_tasks.py
    - utils.py
  - data *

- **Detailed file explanations:**
  - Python files:
    - **app.py**: This file communicates with the backend (server side) and frontend (client side). Flask is the framework used. This file specifies routes to direct the AMT worker to.
      - Each route can have GET or POST methods. GET methods are for accessing the route and POST methods are for sending information to this method. (You must use the <form> tag in the HTML to send information.) For GET, you specify the HTML file you want displayed for the given route. For POST, information is passed to the method as a form, which can be retrieved in python as request.form. (In this project, a list of JavaScript objects is typically sent from the browser through the form. The list of JavaScript objects is converted into a list of python dictionaries:

        ```
        expResults = json.loads(request.form['experimentResults'])
        ```

        The python dictionaries can then be turned into pandas dataframes and saved as csv files. In POST, you can also redirect to another route.
    - **utils.py (inside project folders):** This file contains useful methods to be implemented inside app.py (or the any .py file that handles GET and POST requests for tasks). Methods for generating experimental conditions should be written here. If the conditions for each trial are predetermined, there should be a method that creates a list of python dictionaries where each python dictionary represents the conditions for one trial.
    - **store_data.py:** This file contains methods to write and append to csv files. The methods take a list of python dictionaries (where each dictionary represents the results for one trial) and creates (or adds to) a pandas dataframe, where each row in the dataframe stores information about one trial. The dataframe is then saved as a csv file.
    - **manage_subject_info.py:** The methods in this file write to and read from *_subject_assignment_info.csv and *_subject_worker_ids.csv. *_subject_assignment_info.csv lists the subject IDs and the variables that MTurk passes in, except for the worker ID, which is stored separately in

*_subject_worker_ids.csv. Subject IDs are mapped to the worker IDs through the timestamp (it's possible that two subjects will end up having the same timestamp, but this has only happened once so far…).

- **JavaScript files:**
  - Notes:
    - Multiple JavaScript files can be linked to the same HTML file. For all of the studies, components.js and a script to run the task are linked to the HTML page that displays the task.
  - Variables typically used:
    - expVariables: list of JavaScript objects that stores conditions for each trial
    - allTrials: list of JavaScript objects where each object stores the results/information for one trial
    - currTrialN: index of current trial (1st trial has currTrialN of 0). Used to index expVariables and allTrials to access trial information
  - General structure of some run_task.js (some functions may not be needed, and others will have multiple iterations for a single task)
    - startExperiment(): called in the HTML
    - drawTrialDisplay(): draws all the elements needed for a display (there may be multiple versions of this function for different types of displays). This function should be called on to redraw the screen not only when there's a new trial but also when the browser window is resized (which means that this function should not create any timers). This function will call on pushTrialInfo if added trial information hasn't been added to allTrials yet (pushTrialInfo increases the size of allTrials by 1)

```
if (allTrials.length == currTrialN) {
    pushTrialInfo();
}
```

    - removeTrialDisplay(): clears the screen (clears the canvas and/or removes svg objects) (this function is typically only needed if there are multiple types of displays in a task)
    - pushTrialInfo(): adds trial information to the JavaScript object that stores the experiment's results. Adds current trial information to allTrials. Starts the timer for the trial.

- checkKeyPress() (if you are taking button presses as input): called by

  `window.addEventListener("keypress", checkKeyPress);`

  Records reaction time/any other relevant information.

  Note: `setKeyUp()` and `window.addEventListener("keyup", setKeyUp);`

  are also used to make sure holding down the button isn't registered as multiple button presses
- endTrial(): cleans up the current trial's display, calls nextTrial
- nextTrial(): iterates currTrialN, checks if experiment is over (ex: out of trials or if a time limit was set)
  - if the experiment is over, calls the following code to send the results of the experiment back to the server.

```
var strExpResults=JSON.stringify(allTrials);

document.getElementById('experimentResults').value=strExpResults;

drawLoadingText();

document.getElementById('exp').submit()
```

- resizeWindow(): called by

```
window.onresize = function() {

        resizeWindow();

}
```

- **run_ratingtask.js:** Displays an image at the center of the screen and a rating scale at the bottom, which takes mouse clicks as responses.
- **run_choicetask.js:** Displays two images side by side and a box at the center to acknowledge response or indicate no response.
- **components.js:** This file contains JavaScript classes and methods for different experiment components. In general, changes should not be made to this file. It should be treated like a library of code you can access from other JavaScript files. Right now, it contains classes and functions for creating rating scales, rectangles (which can be used to confirm responses), and text boxes. There's also a function for generating off screen canvases.

- **HTML files:**

        - Notes:
            - To add JavaScript files from the static folder, use

              `<script src="{{ url_for('static', filename='filename.js') }}"></script>`
            - canvas and svg elements are useful for displaying images and objects on the page. The elements can be created in the HTML page. Each can take up the entire window and overlap. Both allow you to position stimuli using coordinates. I use canvases solely for loading images (drawn objects and text in canvases have poor resolution) and svg to draw interactive components on the screen (rating scale, confirmation box). An advantage of using svg is the ability to add listeners to svg objects, making it very easy to determine if, for example, the user's mouse is over the svg object or if the user has clicked on it.

- Notes:
    - Displaying images
        - Images can be displayed on HTML canvases. To make sure images are displayed without delay on each trial, they should be pre-rendered before the experiment starts rather than rendered when each trial is supposed to be displayed. See generateOffScreenCanvases(drawDisplayFunction, name) in components.js. See run_choicetask.js for an example of how images are pre-rendered on canvases that are off screen, and how those canvases are drawn when a trial display needs to be drawn.
    - Timing
        - setTimeout(do_x(), x_milliseconds) can be used to control timing of all events. This creates a timer that calls function do_x after x_milliseconds.
            - Calls to the function are delayed when the window is inactive
                - Effects on the data: if setTimeout is used to end a trial after a certain amount of time and the window is inactive for some duration, the timer may be delayed and the recorded trial durations may exceed the maximum amount by over 500ms. This generally should not be an issue because workers want to finish the tasks as soon as possible. However, this explains why trial durations may exceed the maximum trial duration.
        - performance.now() has variable precision but is more precise than Date.now(). performance.now() returns the amount of time passed since the

document began. Date.now() returns the amount of time passed since January 1, 1970 00:00:00 UTC. (There have been a couple odd cases where performance.now() seems to return what Date.now() returns)

- Testing Web App
  - To test the app locally, go to the terminal and cd into the folder where your web app is located, then run

    ```
    python app.py
    ```

  - You should then see something like:

    ```
    * Running on http://0.0.0.0:8000/ (Press CTRL+C to quit)
    ```

    where 8000 is the port defined in app.run(…) in app.py
  - In a web browser you can then navigate to http://0.0.0.0:8000/ or to http://localhost:8000/
    - In app.py, if you have a route called

      ```
      @app.route("/auction", methods = ["GET","POST"])
      ```

      then that page will be located at http://localhost:8000/auction
  - For convenience during testing, you can set debug = True as a parameter of app.run(…)
    - Doing this allows changes you make in python files to be automatically detected, so after you modify python files, you do not need to restart the app in the terminal – you can just go to the webpage
    - *Make sure to set debug = False when actually deploying the task on the server!*
- Testing JavaScript
  - During testing and debugging, you can print things to the console (using console.log(…)).
  - Every time you're testing JavaScript code, you have do a hard refresh of the web page, otherwise the cached JavaScript file will be used
    - Mac: Cmd + Shift + R
    - Windows: Control + Shift + R
- URLs and MTurk Requirements

- Amazon passes worker information through query strings, which look something like this: https://calkins.psych.columbia.edu/auction_instructions?workerID=abc2286&expName=TEST, where the variables workerID and expName are being passed
- To retrieve these variables from the URL, in app.py, you can use

```
request.args.get('variable_name')
```

Steps for setting up an MTurk study:

1. Get the tasks running in JavaScript (use the sample web app to run and test a JavaScript task)
   i. Write python methods in utils.py to set up the experimental conditions (return the variable expVariables: a list of dictionaries where each dictionary holds the information for one trial)
   ii. Call that method in app.py and pass expVariables to the HTML and then the JavaScript
   iii. Modify the JavaScript as necessary
   iv. The last task should be redirected to the thankyou page, which handles submission of the HIT back to MTurk (see the route for thankyou() in app.py, which requires the argument 'live' to be passed in through the URL)
2. Add routes for the consent form and instructions; connect the tasks together (when request.method == "POST" for task 1, use redirect_url to load the webpage for task 2)
3. Convert the file app.py into a flask blueprint (the directions are specific to adding to this repository, but can be adapted for a different web app as well)
   i. Create a new directory under MTurk that is named after the study (ex: foodchoicestudies)
   ii. Add an empty file named __init__.py to that directory. This allows the directory to be recognized by python as a package.
   iii. Add the flask/python files to that directory. (Only include the methods that aren't already in the app.py in the repository) Rename app.py to a name specific to the study – for a very vague example for the purposes of these instructions, tasks.py. app.py should only have routes for pages that are not experiment-specific (like the error handlers, the thankyou page and the feedback page).

iv. Create a blueprint by making the following changes: for tasks.py, for example, add the following line to the top of the file (after the imports) (and also make sure to have Blueprint imported from flask)

```
tasks = Blueprint('tasks', __name__, url_prefix='/<expId>')
```

v. Change the routes: the routes should look like

```
@tasks.route("/auction", methods = ["GET","POST"])
```

instead of

```
@app.route("/auction", methods = ["GET","POST"])
```

(Specifying the url_prefix makes the url /<expId>/auction instead of just /auction)

vi. In app.py, add and register the new Blueprint. For example –

```
from foodchoicestudies.tasks import tasks

app = Flask(__name__) # should already be in app.py

app.register_blueprint(tasks)
```

4. Test app.py with the new blueprint
5. Add checks for arguments that MTurk passes to the URL
   i. Look at consent_form(expId) in any of the existing studies for examples
   ii. MTurk passes workerId, assignmentId, hitId, and turkSubmitTo as arguments. I also pass live to see if the current HIT is the in the sandbox or not.
6. Do the following tests:
   i. Test the MTurk preview
      i. After loading the consent form page and URL variables are generated, set the assignmentId to ASSIGNMENT_ID_NOT_AVAILABLE. Make sure that only the desired preview can be seen and that the accept_hit page is shown when the preview is over.
   ii. Test the entire task
      i. When the thankyou page is reached, test the following:
         1. When live=True, the next page should have the URL https://www.mturk.com/mturk/externalSubmit
         2. When live=False, the next page should have the URL https://workersandbox.mturk.com/mturk/externalSubmit
7. Check that data are being stored correctly

    i. <expId>_subject_assignment_info.csv and <expId>_subject_worker_ids.csv should be created. Subject data should be stored in individual subject folders.

8. Put the code on the server (make sure app.debug = False).
    i. On calkins, the files are located at MTurk files are located at /projectAlice/MTurk
    ii. Test the code on the server (there may be software incompatibilities sometimes)
9. Restart the server
    i. sudo systemctl daemon-reload
    ii. sudo systemctl restart projectAlice.service
    iii. sudo systemctl restart nginx

Steps for setting up MTurk credentials:

1. Add yourself as an aws user under the lab account, link here
    a. This allows you to create HITs, approve and reject assignments, etc.
2. Install AWS Command Line Interface
3. Configure your terminal with your credentials

Steps for running the study on MTurk:

For more command line options, see:
https://docs.aws.amazon.com/cli/latest/reference/mturk/index.html#cli-aws-mturk

1. Go through the create_hit.py and set all the necessary parameters.
    a. reward amount, url.text (the URL of the experiment; and keep "live=" + str(create_hits_in_live)), masters qualification, and all the variables in create_hit)
    b. See this and this for references for worker requirements.
2. Use create_hit.py to create a HIT in the sandbox and test the HIT in the sandbox. Verify that the preview works as desired and test the entire HIT.
3. create_hit.py uses my_external_question.xml. See this for more information. Note that my_external_question.xml is changed by create_hit.py and passes /?live=True or /?live=False as the url, depending on the parameters, assuming that the home page for the study is at / The element ExternalURL is the url for the webpage that is embedded in MTurk for the HIT. FrameHeight, in pixels, refers to how tall the frame should be.
4. Run the live HIT

   a. Keep in mind that you can run a small sample first and then increase the number of assignments for the HIT (although you should start at a minimum of 10 - if you start with <10 assignments for a HIT, Amazon won't let you increase the number of assignments to >10 because of their 20% fee for HITs with >10 assignments).
   b. Make sure to monitor the email account in case workers run into any issues
5. Use manage_hits.py to list HITs (get the HIT ID), expire HITs, delete HITs, and to list assignments (which gives you a list of worker IDs, their assignment IDs, and the status of their assignments (Submitted/Approved/Rejected))
6. Review assignments
   a. To approve or reject individual assignments, you can use approve_assignment.py and reject_assignment.py
   b. retrieve_and_approve_hits.py approves all assignments that have not yet been reviewed
   c. get_assignments_to_reject.py is an example script that does preliminary analysis of each subject's data and checks whether the subject's assignment should be rejected or not.
7. Set up a dummy HIT if necessary
   a. To compensate workers who were unable to be paid through MTurk (if there's a server error, for example), you can set up a dummy HIT. Create a csv file called dummyHIT_subjects_to_compensate.csv and in the first column, add a list of workers IDs for workers you need to compensate (the title of the column should be subjectId). Go through create_dummy_hit.py to set the payment amount, description, etc., test the dummy HIT, and run it.
8. After running the study:
   a. Back up the data
   b. Make sure that on the server, <expId>_subject_worker_ids.csv gets saved as a password protected file.
      i. Open the csv file in Microsoft Excel
      ii. Go to File -> Passwords... and set a password for the file. (Write the password down! It can’t be recovered.)

Notes:

- Payment:
  - Payment for an assignment is detailed in create_hit.py
  - Go here to add money for prepaid HITs
  - Pricing details are here