

## PA1 — Stateful Tracking on the Web (3 + 4)

**Project Release:** 10/14/2026

**Part 3 Deadline:** 10/22/2026

**Part 4 Deadline:** 11/3/2026

### Introduction

Congratulations! You have successfully implemented a search engine that tracks people, and **Boogle has gone viral!** The company is lauded for its amazing search capabilities and spot-on suggestions, especially that feature where the search results you've done in the past are displayed on the main page. Nice work!

However, nothing in the world comes for free, and so the time has come for Boogle to try and monetize. Rather than make people pay money for the service, the Boogle execs want to leverage their search vantage point and participate in the *online advertising* game. They have specifically chosen **you**, a now fully competent web developer, to implement their ad integration strategy. Your task will be to use Boogle's newfound virality (and data vantage point) to build inferences on users that you can use to engage in the real-time bidding advertising ecosystem.

### Goal

This project will introduce you to two new concepts: drawing inferences on users based on their web behavior, and using those inferences to participate in a real-time bidding advertising protocol. You will build on what you've made in parts 1 + 2, first developing a simple classifier that can predict certain properties about users based on their searches, and next using that classifier to build a *supply-side platform* that can participate in bidding.

### Programming and Infrastructure

This project does not restrict you in terms of the software stack you choose to use. That being said, for part 3, our reference implementation is written in Python and **Scikit-Learn**, a popular and very common modeling library. For Part 4, you will be building on the same stack you used in Parts 1 + 2, unless you want to try your hand at rewriting the entire program or trying a different deployment strategy (which we don't recommend).

### Read this first!

This project asks you to develop technology that can legitimately infringe on people's privacy. Attempting these same kinds of attacks against real people may result in fines, expulsion, and jail time. You must not use these techniques to attack or deanonymize anyone without authorization! You are required to respect the privacy and property rights of others at all times or you will fail the course.

### Forming a Group

This project can be done solo or in a group; you MAY work in a team of size at most two and one entry per team. You are not required to work with the same team on every PA. You and your team should collaborate closely on each part.

The code and other answers you submit must be entirely your team's own work. You may discuss the conceptualization of the project and the meaning of the questions with other teams, but you may not look

at any part of someone else's solution or collaborate with anyone other than your team. You may consult published references, provided that you appropriately cite them (e.g., with program comments).

Solutions must be submitted to Gradescope.

## Index

- [Setup](#)
- [Part 3: Inferring things about users](#)
- [Part 4: Selling ads on Boogle](#)
- [Resources and Guidelines](#)

## Setup

We provide you with some starter code for parts three and four. You can download the starter code here: [http://kumarde.com/webtracking-fa26/starter/starter\\_3+4.zip](http://kumarde.com/webtracking-fa26/starter/starter_3+4.zip).

After unzipping the archive, you should see the following directory structure:

```
starter_3_4/
├── part3/
│   ├── load_data.py
│   ├── train.jsonl.gz
│   ├── train_labels.csv
│   ├── dev.jsonl.gz
│   ├── dev_labels.csv
│   └── test.jsonl.gz
└── part4/
    ├── new_boogle.html
    ├── segments.json
    ├── dsp.py
    ├── bid_request.example.json
    └── bid_response.example.json
```

## Part 3: Inferring things about users

In this part of the assignment, you will be implementing a *classifier* that can infer properties about *synthetic users* based on their search results. We provide you with three files: `train.jsonl.gz`, `dev.jsonl.gz`, and `test.jsonl.gz`, that contain GZipped information collected about the users as well as their searches (one user per JSON line). Here is a part of the first row from `train.jsonl.gz`:

```
{
  "client_id": "eb4cb5f6-17cb-4bc2-8e6b-91860719c93d",
  "first_seen": "2026-06-09T01:07:06.593Z",
  "last_seen": "2026-08-25T23:48:10.507Z",
  "latest_ip": "33.77.57.81",
  "latest_user_agent": "Mozilla/5.0 (X11; CrOS x86_64 1310.0.0)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/124.0.9.1 Safari/537.36",
  "ips": [
    "33.77.57.52",
    "33.77.57.81"
  ],
  "user_agents": [
    "Mozilla/5.0 (X11; CrOS x86_64 1310.0.0)"
  ]
}
```

```

    AppleWebKit/537.36 (KHTML, like Gecko) Chrome/124.0.9.1 Safari/537.36"
  ],
  "accept_languages": [
    "en-US,en;q=0.9"
  ],
  "searches": [
    {
      "query": "cheap meals for college students",
      "ts": "2026-08-25T23:48:10.507Z"
    },
    {
      "query": "how to pick a major",
      "ts": "2026-08-17T03:43:33.701Z"
    },
    ...
  ]
}

```

As you can see, the information in each row is very similar (if not identical) to what you collected in part 2 of the PA.

For each user, your goal is to predict properties for each of *nine* categories: Gender, Age, Education, Household Income, Industry, Employment Status, Employer Size, Home Ownership, and Parental Status. These fields exactly match Google’s *own* taxonomy that they expose via [Google Ad Center](#). The full taxonomy for your labels is provided below:

| Attribute         | #  | Labels                                                                                                                                           |
|-------------------|----|--------------------------------------------------------------------------------------------------------------------------------------------------|
| gender            | 3  | male, female, unknown                                                                                                                            |
| age               | 6  | 18-24, 25-34, 35-44, 45-54, 55-64, 65+                                                                                                           |
| education         | 4  | high_school, some_college, bachelors, graduate                                                                                                   |
| household_income  | 7  | top_10, 11_20, 21_30, 31_40, 41_50, lower_50, unknown                                                                                            |
| industry          | 13 | tech, healthcare, education, finance, retail, construction, hospitality, transport, government, manufacturing, arts_media, other, not_applicable |
| employment_status | 6  | employed_ft, employed_pt, self_employed, unemployed, student, retired                                                                            |
| employer_size     | 5  | solo, small_1_49, mid_50_999, large_1000_plus, not_applicable                                                                                    |
| home_ownership    | 3  | own, rent, other                                                                                                                                 |
| parental_status   | 2  | parent, not_parent                                                                                                                               |

Table 1: Label Taxonomy

We also provide two files, `train_labels.csv`, and `dev_labels.csv`, which provide *ground truth labels* for each of the users who visited Boogle, linked by `client_id`. For example, this is the first line of `train_labels.csv`:

```

eb4cb5f6-17cb-4bc2-8e6b-91860719c93d,male,18-24,some_college,lower_50,not_applicable,
student,not_applicable,rent,parent

```

Note that you should treat **unknown** (when it appears) as a class label, rather than dropping it altogether. Identifying when to pick a certain user as *unknown* is a real use case many advertisers face!

## The pipeline

You should treat this as a simple machine learning task—you will build a classifier that, trained on the available training data, can make predictions about new users *not* in the training data. To that end, you can evaluate how well your model performs using the `dev` dataset, for which you are also provided labels. You should evaluate your classifier using the **Macro-F1 metric**—which weights your decisions across all labels and is used for categorical classification (as opposed to binary classification) in the real world.

Ultimately, your goal is to produce a file, called **predictions.csv**, which has the following header:

```
client_id,gender,age,education,household_income,industry,employment_status,  
employer_size,home_ownership,parental_status
```

Each row should correspond to a row of users from the `test.jsonl.gz` file with your predictions about each user in the testing data. You will be graded based on the Macro-F1 score for each class label for each attribute.

You should note that the objective is **not** a perfect M-F1 for every class. On the contrary, these synthetic users are more like “real people,” and so it’s unlikely you can do exceptionally well on any one particular attribute. The **M-F1 Target** column below shows you M-F1 scores you should achieve should you get full points on the assignment for each attribute, measured against our own reference implementation model.

| Attribute         | Classes | M-F1 Base | M-F1 Target | M-F1 Extra Credit |
|-------------------|---------|-----------|-------------|-------------------|
| gender            | 3       | 0.237     | 0.497       | 0.639             |
| age               | 6       | 0.088     | 0.498       | 0.641             |
| education         | 4       | 0.146     | 0.450       | 0.579             |
| household_income  | 7       | 0.123     | 0.241       | 0.310             |
| industry          | 13      | 0.046     | 0.470       | 0.605             |
| employment_status | 6       | 0.140     | 0.578       | 0.743             |
| employer_size     | 5       | 0.119     | 0.478       | 0.615             |
| home_ownership    | 3       | 0.252     | 0.562       | 0.723             |
| parental_status   | 2       | 0.415     | 0.644       | 0.828             |

Table 2: Score thresholds for each attribute.

For context, our reference implementation uses TF-IDF to vectorize query texts and a simple logistic regression for classification. Our model runs on a simple, commodity CPU and trains in about 10 seconds. If you’d like to do more complicated things, good luck, and hopefully you can achieve the extra credit! :)

## Testing your predictions

We have set up a server at <https://webtracking.sysnet.ucsd.edu/pa1/part3/> for you to test `predictions.csv`.

## Submitting your work

You will submit two files—a `predictions.csv` with test predictions, as well as the code you used to generate your model. Both of these must be submitted to Gradescope. We will not run your modeling code, but we reserve the right to examine it at future dates for academic integrity violations.

## Part 4: Running ads on Boogle!

As Harvey Dent once said, “You either die a hero or live long enough to see yourself become the villain.” It is time to run ads on Boogle.

In this section, you will build a **Supply-Side Platform** (SSP) that enables Boogle to connect to the broader advertising ecosystem and run a real-time auction for an advertising slot on the website. You will combine your web server from parts 1 + 2 and your inference from part 3 to place a *targeted advertisement* on your website by running a real-time auction. In the process, you will also have to handle *complaints* about the advertisements you place, as advertisers may have specific targeting requirements, or sometimes, advertisers might want to place unsavory or suspicious ads. Ultimately, your goal is to make a reasonable net profit for Boogle.

## Getting started

To begin, you will need to swap the `boogle.html` file from before to `new_boogle.html`, which contains a single HTML file that contains a 300x250px advertisement inside of an `iframe`.

To implement your SSP, you will need to add two new routes to your server: `/ad`, and `/ad/feedback`. We detail each below:

## Implementing the auction: GET `/ad`

The included `iframe` automatically makes a GET request to `/ad` when it is loaded. This route should run a *single* auction for the user making the request (e.g., the one whose `boogle_id` is stored in the browser).

**Making requests to DSPs.** Your SSP must communicate with Demand-Side Platforms (DSPs) we have set up, at the base URL <https://webtracking.sysnet.ucsd.edu/pa1/rtb/>. We have four DSPs, a “steady” DSP that bids regularly on most ads, a “niche” DSP that pays **only** for select user segments, a “shady” DSP that occasionally produces bad advertisements but gives lots of money, and a “glitch” DSP that occasionally presents malformed requests. The DSPs each live at the following routes (relative to the base URL): `/steady/bid`, `/niche/bid`, `/shady/bid`, and `/glitch/bid`.

When running an auction, you should make an OpenRTB request to all four DSPs and see what you get in reply. In particular, you should send an HTTP POST to each route, with data of the following form:

```
{
  "id": "6c1f0e0a9d6b4f51",
  "imp": [{ "id": "1",
    "banner": { "format": [{ "w": 300, "h": 250 } ] },
    "bidfloor": <UP_to_you> },
  "device": { "ua": "Mozilla/5.0 (iPhone; ...", "ip": "137.110.222.15" },
  "user": { "id": "<the visitor's boogle_id cookie>",
    "data": [{ "id": "boogle",
      "segment": [<the_user_segements_you_claim>],
      "ext": { "segtax": 500 } } ] },
  "tmax": 250,
  "badv": []
}
```

We provide a full example request JSON in `bid_request.example.json`. This object is a subset of the [IAB's OpenRTB specification](#)—you can check the reference there for more details on each of the fields we provide (as well as the ones we highlight in lecture).

Your request should contain *user segments*—in the `user.data` portion of the request. These segments should be inferred based on the *active searches of the current user*, using exactly your model from part 3. We provide a mapping from what you inferred in Part 3 to segment IDs in the included `segments.json`. **Unlike part 3, if you do know something definitive about the user, then you should leave their segment out.**

Make sure you include the `ext` field with each request.

**Receiving responses.** Each DSP will respond either with a status code of 204, which means “No bid,” or 200, which contains associated data. As an example, the JSON returned will have the form:

```
{ "id": "6c1f0e0a9d6b4f51", "cur": "USD",
  "seatbid": [{ "seat": "steady", "bid": [{
    "id": "b-1d2c...", "impid": "1", "price": 1.84,
    "adm": "<div data-boogle-ad ...>...</div>",
    "adomain": ["brightdeals-1a2b.example"], "crid": "cr-9f8e...", "attr": [12],
    "w": 300, "h": 250, "mtype": 1,
```

```
"burl": "<RTB_URL>/bill/b-1d2c...?price=${AUCTION_PRICE}"]]]}
```

In particular, note that the advertisement itself (i.e., what you load in the `iframe`) is located in the `adm` field in the JSON response. Also note that a response may contain *several* bids from the same DSP—you should consider each bid when making your final choice.

**Filtering bad bids.** Sometimes, our advertising DSPs will make malformed or otherwise bad bids. You need to implement some logic to filter these out. In particular, your SSP should filter bids that:

1. Arrive after `tmax`
2. Are not valid JSON
3. Is in a currency other than USD
4. Names an `impid` you didn't send
5. Is not for a 300x250 slot
6. Is below your bid floor

All of these checks will be exercised by the autograder.

**Picking and pricing the winner.** Of all the valid bids, you finally need to decide which ad to deliver to the page. The highest bidder wins, but pays using the following algorithm:

$\text{price} = \min(\text{winner's bid}, \max(\text{second-highest bid}, \text{your floor}) + 0.01)$

This is known as a **second-price auction**, and is very common in real-time auctions to try and promote “true, fair” bidding. In essence, you are simply charging 0.01 more than the second-highest bidder, to the entity that bid the highest amount.

**Note:** There may be a scenario where no one bids on the ad. In this case, you should return a 200 status code and display nothing in the `iframe`.

**Billing the DSP.** Finally, you should make a `GET` request to the `burl` in the response, with the `AUCTION_PRICE` field set to your final price. This is how we keep track of what money Google actually made from this exercise.

## Handling complaints: `/ad/feedback`

Google will only earn what a biller ultimately pays, and one ad at a price of 2.00 Cost per Mille (CPM, in the RTB response) earns Google \$0.002.

Unfortunately, sometimes, users and advertisers alike complain about the ad placement, causing your revenue to take a hit. You will receive `POST` requests, occasionally, on `/ad/feedback` which contain these complaints. The data will be in the form:

```
{"auction_id": "6c1f0e0a9d6b4f51", "event": "complaint"}
```

Where the `auction_id` is what you sent to our DSPs. Many of these complaints may come from high bids, so you will need to additionally keep track of which advertisements **or** advertisers continue to draw complaints. You can either drop them entirely, or include those advertiser domains in the `badv` field in your request, and our DSPs will honor those faithfully.

Ultimately, your net profit is equal to your revenue minus complaints.

## How your system should work end-to-end

Your system should serve the Google page, run the real-time bid, and display the winning advertisement to the page for all users. Our autograder works as follows:

1. We create a pool of N synthetic people with our DSPs, drawn from the same distribution as part 3
2. Each “person” makes searches, so your database will eventually fill up with searches
3. Each “person” looks at two ads by directly calling `GET /ad`
4. After each round, every complaint is sent to `POST /ad/feedback`
5. We run this dozens of times for each grading run

On top of checks that simply identify whether you implemented the logic, there are three additional meta-checks we do for the full grading run, which are:

- Google actually makes money—net profit should be above zero
- The niche DSP wins and pays for the ads (so your model is hooked up correctly)
- Ads that eventually cost more because of complaints stop winning

## Testing your SSP

We provide a local file, `dsp.py`, that can run locally and “emulate” one of our DSPs. The code is very straightforward, with all DSPs simply bidding 2.00 CPM for each ad. This is basically a mechanism that you can use to test locally to make sure your logic is doing what you want and the ad being returned is properly displayed.

When you are ready to test your system in full, we have set up automated infrastructure on live DSPs at <https://webtracking.sysnet.ucsd.edu/pa1/part4>. Similarly to parts 1 + 2, we require a public IP address that can accept connections at all routes. The same rate limits apply to this autograder, so be choosy with your submissions.

## Submitting your work

You will submit a single file, called `server.txt`, to Gradescope as your final submission. The file will contain a single line—with the form `< ip_address >:< port >`. Note that your server **must be up** and running consistently when our grader runs (which will be some time after the project is due).

## Resources and Guidelines

This project requires you to get familiar with a lot of different technologies, so here are some resources that can help you along.

### TF-IDF x Logistic Regression

Our reference implementation for part 3 uses a simple TF-IDF x Logistic Regression combination. There are lots of tutorials online to explore these ideas, but you might want to look here: [to start](#).

### SSPs and DSPs

Amazon has a lot of great resources on online advertising—including [this great primer](#) (and jumping off point) to really understand the nitty gritty of demand side platforms and supply side platforms.

### OpenRTB

There is no better place to really deeply understand something than to look at the specification itself, so I highly recommend you at least scan through the [OpenRTB 2.6 standard document](#).