CS 590 HW 3: Automated Mechanism Design
Nicholas von Turkovich

Relevant files:
on_paper.pdf
hw3.mod
analyze.py
test.py
mechanism.txt

Introduction

In this homework, we were given a stream of (potentially) correlated bid-signal pairs with the objective of designing a mechanism that maximized revenue when allocating an item.

This assignment was completed in 5 steps:

1. Write the linear constraints and objective function down on paper
    a. BNE-IC, EX-POST IR, EX-INTERIM IR
2. Translate these linear constraints to a .mod file to solve for the optimal allocation probabilities and payments
    a. Supply joint and conditional distributions as parameter fields
3. Write a .py file to load the dataset, parse it, create training and testing sets, compile a joint distribution and conditional distributions based on valuation type
    a. The formatted output could then be copy/pasted into the .mod file
4. Write and use a .py file for testing purposes, which calculates for each test bid-signal combination, what the different utilities for reporting different types using the full distribution (training and testing)
    a. The agent then tries to maximize its value
    b. Based on the action taken, calculate the payment and average to get a testing revenue estimation

On Paper

The first step to this assignment involved playing with a marginally less complicated version of the problem and writing down the linear constraints and objective function. See on_paper.pdf for the notes.

Linear Programming with GNU Kit

The notes were translated into a LP in hw3.mod. The sections of the code are broken down by comments. First, the objective is to maximize the expected revenue given the prior (believed from data) joint distribution. Note that the joint distribution here is calculated in a .py file (which is mentioned later) from the set of training data. Second, there are 3 linear constraints. Two of them refer to the different individual rationality constraints. Both were tested but the constraint that produced the best results was ex-interim IR. The third restraint is for BNE IC.

Finally, in data, there are two parameters for prior_probs (joint) and prior_prob_conditionals (conditional on the agent valuation). Note that the values in these parameters are calculated in analyze.py and those left in there at submission may be an artifact of a test.

Generating Training/Testing Sets and Compiling Distributions

In analyze.py, the text file with training data is read, randomized, and split into testing/training sets (70-30 split). From the training data, a frequency table is created and then normalized to get the joint. From the joint, a table with conditional distributions is also made. These two distributions are used in hw3.mod.

The full data distributions were also compiled and saved for testing later.

Commented out, at the bottom, is a visualization module for observing the distribution of the data across its two variables.

Testing

After running glpsol, the allocation probabilities and payments were loaded into test.py. This file uses the testing data to simulate how a bidder would operate using the schema. All expected utilities are calculated for each reported type using the full distribution. The agent takes the max and determines which action will lead to that max. If the action is to not participate (utility = 0), then revenue for that agent is 0. Otherwise, the payment for the realized reported type and signal is added to a cumulative revenue value which is later averaged over the number of

bidders. This testing scheme could be run for different subsets of the population to check for robustness.

Discussion

A couple notes about how I arrived at the allocated probabilities and payment matrices. First, under the ex-interim constraint, the solver sets all of the allocation probabilities to 1. This intuitively makes sense as there is no detriment to the seller given the prompt and it can only increase the expected utility of each bidder.

Second, the payment matrix is not the result of a training/test set. It's the average of a number of different matrices, each performing well with different subsets. To clarify, I began by running the simulation until I reached a test value that was relatively high. I inferred that this training iteration did well since it was able to get a good 'breadth' of the data points, hence a good idea of the distribution (not overfitting to a small pocket of data). From there, I ran the testing on that matrix over and over with different testing data to find a dataset that the current payment matrix "missed." Once a bad testing scenario came about, the weights for that testing/training set combination were calculated and averaged with the previous 'relatively good' payment matrix.

The rationale for doing this was that in every test, one can only hope for doing generally well with the data. However, there will be some deficiency. One payment matrix may do well for a testing set with a particular swathe of the distribution. By iteratively checking for datasets that discovered deficiencies, calculating those training sets' payment matrix, averaging all of the 'allowed matrices', and rechecking the performance, the payment matrix performance got incrementally better. (all testing average revenues between ~2.8 and 3.3)