Scheduling Tasks under Constraints

CS229 Final Project

Mike Yu
Stanford University
California, 94305
myu3@stanford.edu

Dennis Xu
Stanford University
California, 94305
dennisx@stanford.edu

Kevin Moody
Stanford University
California, 94305
kmoody@stanford.edu

Abstract

The project is based on the principle of unconventional constraints on schedules. Generally, a student's calendar is not so constrained that they must choose which assignments are completed and which ones are left unfinished. Rather, it is the issue of procrastination that necessitates schedules for most students. Fighting the temptation of procrastination is very distinct among different people; some enjoy brief periods of work with intermittent breaks, some frontload the work to get it over with, and almost everyone wants to have a regular sleep schedule that's uninterrupted by their assignments. On this basis, we seek a way to evaluate schedules based these tendencies, aka the desire for chunking the work together, or regularly abstaining from work on Friday nights, etc. The premise of the problem is that, given all the work the user is exerting over the week, the algorithm can generate a schedule that most accurately fits their personal desires for how often and when to work.

1 Introduction

The productivity space today is rife with applications that allow you to account for what you need to do and when it gets done. However, nothing tells you when to do it. We believe that the main reason for this is that each person has personal preferences for when and how they like to approach tasks, and this makes it difficult for a standard program to schedule tasks for individuals. However, we believe that instructing people on when to complete tasks, tailored to an individual, can be done, as long as we learn each persons individual preferences, and with that information, personalize recommendations.

To this end, we model a person's preferences with a "cost" function for a given scheduling, using features extracted from the scheduling. We use data about a person's past schedulings, as well as the associated cost, as input to stochastic gradient descent to "learn" an individual's cost function, and then use this function to solve uniform cost search problems taking in new tasks, telling the user when to perform these tasks.

2 Datasets

2.1 Obtaining Data

Before beginning the algorithm, numerous random tasks are generated, as data. There is no need to be overly restrictive about what random tasks we may need to deal with, as the lengths, start times and end times of assignments tend to vary wildly in the real world. The only requirement on the generation of these is, of course, that one has enough time between the start and deadline to actually complete a task, as impossible tasks offer no useful data. These are the training data "tasks". We then generate the training data "schedules" based on what we perceive as a reasonable reward function for a person to have, based on the uniform cost search problem illustrated below. Finally, we use our ideal cost function to generate the cost of this schedule. This data needs to be generated for two reasons:

1. Our algorithm is currently restricted to taking in a cost associated with the task. These do not exist in practice, and while we have ideas to remove the need for these, learning a function where you don't know the value, but rather the schedule that minimizes the function over a space, is as of now an open academic question for which no algorithm has been established. It is something we have made numerous forays into (start with an estimate and iterate the weights based on the gradient of the weights at the point given by

the current schedule), and still have some ideas on (leave one data point out, compute weights with other points, find that data point's "cost" by the learned weights of other points, recurse down to base case (at which point we guess), repeat for all points) which are computationally infeasible, we haven't been able to make meaningful headway on this mathematically tough problem

2. That much data simply isn't kept by a single person, as far as we know

2.2 Data - Schedules

The table is a sample of 10 training data points (tasks and how they were scheduled under the ideal cost function); the full sets of 100 data points, training and test, are attached and were not listed here for space reasons.

Table 1: Training Data

Task Triple	Schedule Under Ideal Cost
$(4, 133, 100)$	$[103, 110, 120, 127]$
$(4, 161, 120)$	$[123, 129, 136, 146]$
$(4, 158, 120)$	$[123, 129, 136, 146]$
$(4, 81, 76)$	$[77, 78, 79, 80]$
$(4, 126, 78)$	$[81, 87, 95, 106]$
$(4, 59, 13)$	$[16, 23, 31, 42]$
$(4, 61, 35)$	$[37, 42, 47, 54]$
$(4, 109, 80)$	$[82, 86, 90, 95]$
$(4, 156, 139)$	$[141, 144, 148, 153]$
$(4, 59, 20)$	$[23, 29, 36, 46]$

3 Features

After experimenting with numerous sets of features, we selected these five features, for what we believed to be a fair representation of a task and the costs associated, as well as to prevent overfitting and increase the chances of convergence:

- ϕ_1 = how late the bulk of task is completed (productivity under pressure)
- ϕ_2 = how long to wait before starting (procrastination)
- ϕ_3 = sparseness of the task completion set (chunking tendencies)
- ϕ_4 = how early task is completely finished (stress tolerance)
- ϕ_5 = how many hours worked on Friday (blacking out a specific day of the week)

Each of the above features provides an important piece of information regarding the cost of a user working for an overall schedule s_i. While we experimented with other features (such as one for each day of the week, and also a set for times of day), the feature set we selected provides an accurate sample for the purposes of academic demonstration, and more importantly, converges with ease; adding the 12 or so features needed for times of day as well as days of week caused overfitting problems (due to the sparse nature of data in each group).

4 Model Selection and Implementation

4.1 Model Selection

In general, our goal is to create a function where the input is a 3-tuple "task" t, where $t = (n, b, a)$, and n is the size of the task (in # of hours), and a and b are the start and end (deadline) times of the task, respectively, as measured by 0-indexed hours of the week. These three integers will have domain [0 - 167], to denote all available discrete hours in one week. Ultimately, the desired output will be a "schedule" array of the same size as the inputted

task size, describing the 0-indexed hours of the week the task will be completed; we index the week with 0 being midnight on Monday morning.

As noted in the introduction, this function needs to be unique for each person, as it must reflect an individual's preference. Then, it is generated with a machine learning algorithm, which takes in a series of past tasks completed by the user, along with the schedules which they were completed under, and the "cost" of that schedule (ideally, in the future this cost could be removed from the necessary inputs, as in practice these are difficult to obtain). This data of past tasks is used to rederive a user's cost function, which can then be used in our scheduling function to schedule new tasks.

4.2 Using Uniform Cost Search

The uniform cost search we use to determine the optimal schedule using our heuristic rewards function also represents the oracle stage; as this is exactly what our "artificial user" would like. The states in this state space are described as State $= (t, s, i)$, where t is the task, s is the array of schedules, and i is the index-hour of the week currently being observed, where $i \in [0, 167]$, with 0 corresponding to midnight Monday morning, as previously described.

An action consists of either adding the hour to the scheduled work times, or not, so the possible action space at each state is $\text{Actions}_{\text{state}} = \{\text{add}, \text{don't add}\}$. The transition for "add" at a given state increments index-hour by 1 and adds the index-hour to the schedule-array; the transition for "don't add" also increments index-hour by 1, but doesn't add the index-hour to the array, so, formally, we have that $\text{Transition}_{\text{state, add}} = (t, s + [i], i + 1)$, $\text{Transition}_{\text{state, don't add}} = (t, s, i + 1)$.

Finally, we need a start state and end conditions. The start state consists of an empty scheduling array, and we begin looking for possible hours at the start time, orStart $= (n, b, a), [\,], a)$, and we are done if we've either fully scheduled the task, or hit the deadline. The end conditions are then either $|s| = n$ OR $i > b$.

We use a cost function to evaluate the schedules; either the ideal function (in the case of generating training data) or our learned experimental function (if we are operating on test data). The start state will consist of (task 3-tuple, [], task-start-time), and the end conditions are "there are no more hours left in the task (scheduling array has reached size equal to task size)" or "index-hour has reached task-end-time and the week is over".

4.3 Learning a Cost Function

An "ideal" set was weights for the five features listed above was chosen to generate our training data, complete with schedules and associated costs. Because the cost function is in fact a weighted feature extractor, we can use this training data (schedules and costs) and determine, using gradient descent, a set of weights that fits the data. To perform this, we used batch gradient descent for regression, minimizing error of $\sum_{s \in S}(c - w \cdot (\phi(s)))^2$ for the training data set S where $\phi(s)$ is the feature vector described above, for the schedule s. Then, the batch update is:

$$w \leftarrow w - \frac{1}{|S|} \sum_{s \in S} (c - w \cdot (\phi(s)) * \phi(s)$$

and this is how we learn our experimental set of weights.

4.4 Scheduling New Tasks

Now, equipped with this experimental cost formula, we create a scheduling function. This function uses the same uniform cost search with this new cost formula to determine optimal schedules. In this case, we test not only on the training data tasks, but also on general test data. Note that we can also use the same uniform cost search to determine our baseline, but with a trivial reward function that has 1 for all its weights.

5 Results and Discussion

5.1 Ideal vs. Learned Weight Vectors

Our ideal weight vector, w^* was chosen as:

$$w^* = [1, 1, 3, 1, 10]$$

and our experimental weight vector w, after training on the training data set, was determined to be, with rounding:

$w \approx [1.547, 0.157, 3.685, 1.339, 2.211]$

5.2 Test Results

The table is a samples of 10 ten test data points, and how they were scheduled under the ideal function, as well as under our learned cost function.

Table 2: Test Data Output

Task Triple	Schedule Under Ideal Cost	Schedule Under Learned Cost
$(4, 120, 106)$	$[108, 111, 114, 118]$	$[107, 110, 113, 117]$
$(4, 38, 5)$	$[8, 13, 19, 28]$	$[6, 11, 17, 25]$
$(4, 58, 32)$	$[34, 39, 44, 51]$	$[33, 37, 42, 49]$
$(4, 27, 21)$	$[22, 23, 24, 26]$	$[21, 22, 24, 26]$
$(4, 45, 27)$	$[29, 32, 36, 41]$	$[28, 31, 35, 40]$
$(4, 18, 5)$	$[6, 9, 12, 16]$	$[6, 9, 12, 15]$
$(4, 79, 54)$	$[56, 61, 66, 73]$	$[55, 59, 64, 71]$
$(4, 112, 80)$	$[82, 86, 90, 95]$	$[81, 86, 92, 100]$
$(4, 154, 139)$	$[141, 144, 147, 151]$	$[140, 143, 146, 150]$
$(4, 79, 62)$	$[64, 67, 71, 76]$	$[63, 66, 70, 75]$

5.3 Evaluation

All methods of evaluation were done on both training and test data.

We have several methods of evaluation. In the first two steps of the algorithm, we are essentially determining the compatibility of stochastic gradient descent and uniform cost search. In other words, we will compare our experimental cost function (f) against the pre-determined, heuristic cost function (f^*). In order to make this comparison, we can simply use the old and new cost functions to evaluate the costs of a set of schedules (the set S), and compute the average percent error between the two costs, expressed by

$$\text{Normalized Cost Error} = \frac{1}{|S|} \sum_{s \in S} \left| \frac{f^*(s) - f(s)}{(f^*(s) + f(s))/2} \right|$$

In which we find the average percent error. Using this data, we get that:

$$\text{Normalized Cost Error}_{training} = 0.342107688432$$

$$\text{Normalized Cost Error}_{test} = 0.332337197763$$

This is closely tied to the basic error we are trying to minimize with our gradient descent, and thus should not be very "high", but this seems to show that our cost is not particularly accurate, as over 100 data points, this indicates an average percent error of about 33.2%. Fortunately, this only tells us about the accuracy of our arbitrary cost, which is not the overall goal of the project, which is learning user preferences, and the error can largely be explained by our next method of evaluation, weight vector error.

We could also look to compare the weight vector directly; that is, for our experimental weight vector w, and ideal weight vector w^*, we could simply find the difference between each weight and square it. This error would look something like:

$$\text{Weight Vector Error} = \sum_{i=1}^{n} |w_i^* - w_i| = 10.2022636315$$

There are five weights, meaning that on average, the experimental weight differs from the ideal by $\frac{10.2}{5} \approx 2.04$. This is a pretty large difference, considering the low value of our weights, and this can mostly be attributed to the

$w_5^* = 10$; while in learned error, $w_5 \approx 2.2$, which gives us the error for w_5 alone being about 7.8. This probably arises because not every schedule contains a Friday, as hours 96 to 120 are not necessarily even within the bound for a task, so this makes a lot of sense; a larger dataset might go some way towards mitigating this. In addition, because the schedules outputted are "optimal" and thus minimize cost, such a large weight is likely to be avoided, which means that this feature will almost always have value very close to 0, and so the learning of its weight is less likely to reach its true value.

It is important to note, however, that the closeness of these functions, although ideal, is not necessarily a pre-requisite to good schedules. It may be the case that although the weights are somewhat different, these cost functions nonetheless generate similar schedules under the CSPs; this is especially pertinent when we consider that some of the features are correlated (such as the distance from start to first working hour, and distance from end to last working hour). The goal of this algorithm is to produce schedules that users are happy with, not necessarily derive their rewards function. This leads us to the second mechanism of evaluation, in which we evaluate uniform cost search using a set of tasks S, and do this for each cost function, then compare the resulting optimal schedules. The schedule produced by the experimentally learned function, given task s_i, is x_i, a vector of x_{ji}s, and the schedule produced by pre-determined, ideal function, given the same task s_i, is x_i^*, a vector of x_{ji}^*s. Then, we might measure schedule based error as follows:

$$\text{Schedule Error} = \frac{1}{|S|} \sum_{s_i \in S} \sum_{j=1}^{5} \left| x_{ji}^* - x_{ji} \right|$$

Our test data produces:

$$\text{Schedule Error}_{training} = 8.11$$

$$\text{Schedule Error}_{test} = 6.96$$

Considering that our data was tested on tasks of size 4, this indicates that our average "off by" for each slot is around 2 hours. That's pretty solid, and an indicator that our algorithm was able to fairly accurately predict schedules for a user. This was possible despite the difficulty with properly weighting Fridays (due to the sparseness of their existence in training set schedules, since they are so heavily avoided by our "ideal user"), probably because the relative weights of the other features were largely preserved.

It's also worth noting that in general, our test error is lower than our training error (slightly); this indicates that our formula was definitely not overfit with respect to the training data, which makes sense, as this was something we specifically tried to avoid when not over choosing features (i.e. putting a weight on each possible hour, for instance).

6 Conclusion

In conclusion, while we were able to accurately schedule tasks, provided with enough data, we were ultimately unsatisfied by the need to include ideal cost with the training data as a tool to help us learn a user's preferences. This leaves a lot to be desired, and a lot of space for future work if we ultimately would like this to be a practical and usable algorithm. However, this was an excellent exercise in using various tools explored this quarter, including state space search and gradient descent. We also feel that significant progress was made on the problem, even if we couldn't achieve the crucial breakthroughs needed to make this algorithm useful for any more than academic exercise.

7 Future Work

In most practical applications, data will come without the cost parameters attached. For this algorithm to work for these cases, we need to model cost based on the scheduled times and learn the cost function in that manner. We believe we might pursue this with hold-one-out-cross-validation (computationally expensive) or directly link each input tuple to an individual hour scheduled given enough data (that is, instead of trying to solve the cost function for some "cost" intermediary, instead try to write n uniform cost search problems, where n is the size of the task, and solve these n problems to find n timeslots in which to perform the task; however, this doesn't allow us to take into account sparseness of task completion, which we hypothesize to be an important feature).