Dynamic Filter: Adaptive Query Processing with the Crowd

Doren Lan Beth Trushkowsky

Harvey Mudd College  
dlan@hmc.edu, beth@cs.hmc.edu

Abstract

One advantage of using computers over humans is the speed at which computing retrieves and parses through data. However, computers currently lack some valuable qualities that people posses, including intuition. Computers have trouble answering ambiguous or complex queries without previous data. The motivation behind this project lies in leveraging the speed of computing with the intuition of the crowd. More specifically, computing with the crowd can help with the filter operation, which takes a set of items and passes them through a set of restrictions called predicates. An effective way of performing a subjective filter is leveraging the power of the crowd to sort through the dataset. Our aim is to make filtering using the crowd as efficient as possible, minimizing the number of tasks assigned to workers. We developed an algorithm that adapts to the dataset and passes items to dynamically ordered predicates. This algorithm was tested on synthetic data and data crowd-sourced from Amazon Mechanical Turk.

1 Introduction

Filtering is a common database operation that takes in a list of items and outputs items that pass certain restrictions specified in the filter called predicates. Predicates can be thought of as true/false questions that only allow items that are true to continue through the filter. As an example, a filter operation may look like, given a list of restaurants, filter those that serve Chinese food, have a price range between $11-$20, and have their own website. The list of predicates that the list of restaurants must pass through can be arranged as a series of true/false questions:

Does this restaurant serve Chinese Food?

Does this restaurant have a price range between $11-$20?

Does this restaurant have its own website?

However, a problem arises when the predicate becomes subjective or ambiguous. For example, the question, “Does this restaurant have a romantic atmosphere?” has a more subjective and ambiguous nature that cannot be easily determined by a machine. Instead, the crowd may have a better understanding of what a romantic atmosphere means and whether or not a particular restaurant has this property. Thus, this filtering task is better suited for people to answer. In this case, human workers utilized as a powerful resource [1].

This project primarily used Amazon Mechanical Turk (AMT), a crowdsourcing platform, to gather data on answers to predicates. The basic function of this platform is used for creating and gathering information from micro-tasks, or small units of work that are completed by humans. From AMT, we were able to post Human Intelligence Tasks (HITs) that were made up of an item/predicate combinations for a worker to answer true or false. The answers were recorded and pulled off in batch files to be read by the database as “real data”. For our experiments, we collected two batches of real data. One batch was 5 different questions about 90 hotels, and the other was a smaller batch with 10 questions about 20 restaurants. To ensure the accuracy of the answers, multiple workers answered each item/predicate combination.

In this paper, we will discuss specific functionalities in the algorithm we used to efficiently filter items using the crowd and how we tested this algorithm using real and synthesized data. We will then discuss what the tests imply and the future worker of this project.

2 Predicate Properties

Each predicate has specific properties that may contribute to the efficiency of filtering. Predicates should be ordered in the filter based on these properties to create optimal efficiency. One of these properties is the selectivity of the predicate, or how often the predicate returns false. Suppose that there are two predicates, P0 and P1, in the filter. If P0 has a higher selectivity than P1 with all other things equal, P0 should be asked before P1. This is because more items will return false when passed through P0, so P1 will not have to be asked as often. Thus, the ordering of predicates based on selectivity will change the efficiency of the filter but not content of the final filtered list.

In addition to selectivity, there is a cost associated with each predicate. Each predicate has varying amounts of ambiguity and subjectivity that may require more answers to reach a consensus. We define the average cost of a predicate to be the average number of votes to reach consensus on an answer. The ordering of predicates based on cost also affects the efficiency of the filter as well. Given predicates P0 and P1, where P0 has a lower cost than P1, the efficient ordering would be P0 before P1 if all other things were equal. Conducting cheap predicates first will pass a filtered list to the next predicate with less expense, which may avoid the more expensive predicate from being asked more than necessary. The reordering of these predicates based on cost does not change the output list, but does affect the efficiency of the filter.

3 Task Allocation and Aggregation

An important component of this project is using the crowd to answer predicates. In order to simulate a filter process, we also needed to simulate worker tasks. A single simulation run consisted of this workflow:

|  |
| --- |
| **while** items.notFinished:  worker = chooseWorker()  item, predicate = algorithm()  worker.doTask(item, predicate)  numTask += 1 **return** numTask |

As seen in the pseudo code, each task is assigned by choosing a worker, picking an item and predicate that the worker has not completed yet based on the algorithm, and then allowing the worker to cast a vote for that item/predicate pair. Simulations were run using real data pulled from AMT. In order to simulate a response for a given item/predicate pair, a worker’s answer was generated by randomly choosing from the answers provided by the real data.

While the crowd can be used to answer complex predicates, we must use multiple answers, or votes, to be more confident in answer to the subjective predicate. For each item/predicate pair evaluated by the simulated crowd takes in a minimum number of votes to determine whether the item is true or false for the predicate. If there is not a clear consensus, then two more votes are casted and the process is repeated until a consensus is reached. The certainty of the vote aggregation is based on a beta distribution where the certainty level is equal to the cumulative probability of the combination of true and false votes. The simulation had a cut off number of votes so that the voting for a particularly subjective item/predicate pair will not continue indefinitely.

4 Dynamic Queue Algorithm

As stated in Section 2, the order of predicates based on selectivity and cost may decrease the number of required tasks. However, predicting the selectivity and cost of a given predicate with no prior information may be difficult. This ordering of predicates becomes a multi-armed bandit problem where we have to decide to explore the properties of the predicates to find the optimal filter or exploit a specific predicate to minimize the number of required tasks [2]. A dynamic filter that adapts to the given predicates may be more ideal than a static order of predicates. The algorithm should route items to predicates of higher selectivity and lower cost. The filtering process can be visualized using a river eddy analogy [3]. The items act like water that flow though the filter where each predicate operation acts like an eddy in the river. The idea of the dynamic filter is to have items naturally flow to predicates of lower cost and higher selectivity while avoiding predicates of higher cost and lower selectivity. The following sections will describe how we implemented proxies for these properties so that our algorithm will dynamically route items to desired predicates.

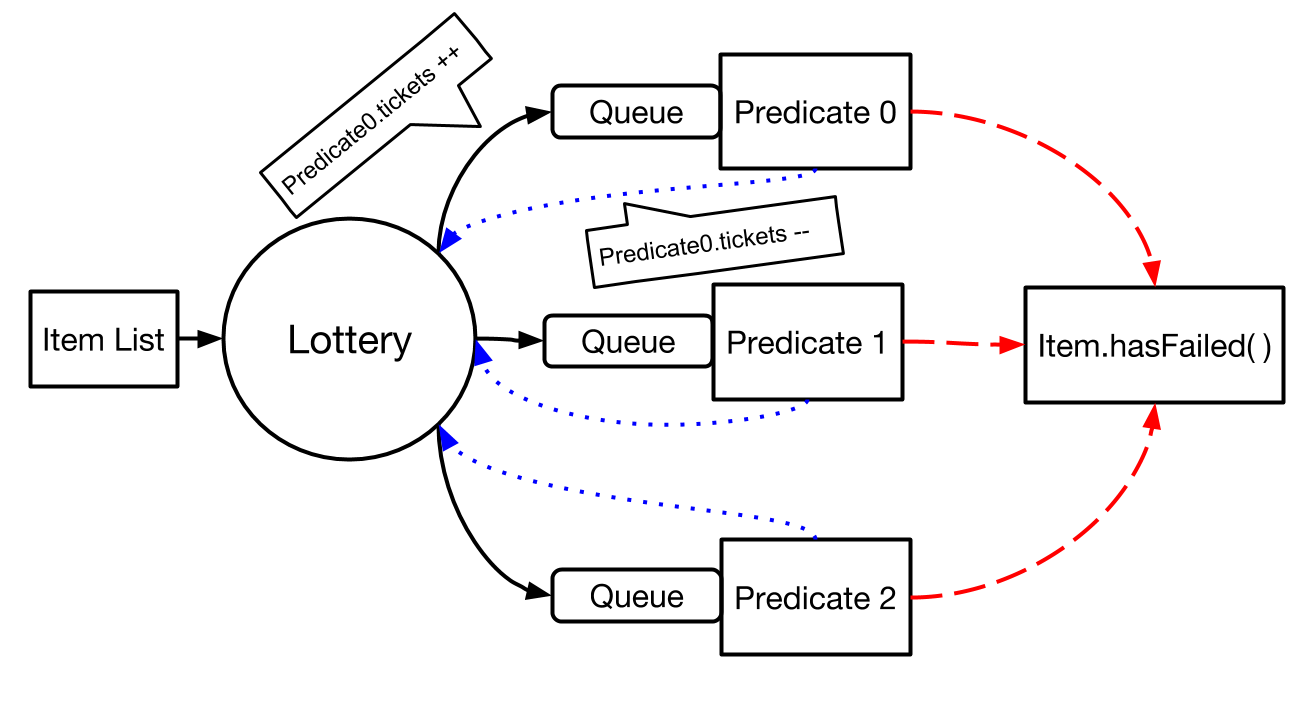
**4.1 Accounting for Selectivity**

To dynamically determine which predicate is most selective, the eddy runs a lottery every time an item needs to be routed to a predicate. The lottery is weighted based on the number of tickets each predicate has. The predicate is given a ticket every time an item is sent to the predicate by the lottery. Once the item is evaluated in the predicate, the item is either true or false for that predicate. If the item returns true, it is sent back into the lottery to be assigned to another predicate and the number of tickets for that predicate is decremented. However, if the item returns false, the item is listed as a false item and is no longer evaluated. The predicate retains the ticket it received for the item. As a result, the more a predicate returns false, the more tickets it receives over time and consequently, more items are routed to this predicate.

**4.2 Accounting for Cost**

While a predicate may be selective, it may also be associated with a high cost. Predicates with high cost are best avoided because the number of tasks require to complete items sent to this predicate will also be high. To avoid constantly routing items to a high cost predicate, each predicate contains a queue that holds items that are waiting to be evaluated by the predicate. While the queue is full, items are no longer routed to that predicate. The more expensive a predicate is, the more tasks it takes to evaluate the item passed to the predicate. Thus, the queue will remain full longer reducing the number of items that are routed to that predicate. These queues ultimately create a “back-pressure” that prevents items from being routed to more expensive predicates. Predicates that require more votes to reach consensus will experience a greater backpressure of items.

In combination with the ticketing system, the algorithm will begin to favor cheaper predicates over time because these predicates receive more items into their queues, which means they have a higher chance of retaining more tickets.



*Figure 4.1: A diagram of the ticketing queue algorithm. The dotted blue arrows represents when the item passes the predicate, and the red dashed lines indicate where the item is routed when it fails*

**4.3 Dynamic Response to Fluctuations**

When dynamically filtering items, the algorithm should be able to adapt to fluctuations in the selectivity or cost over time [4]. If the algorithm only used ticketing and queue systems, a specific predicate may become heavily weighted through the filtering process. As a result, the algorithm will have difficulty adapting to fluctuations in the selectivity or cost of the predicates in the filter. To implement a more adaptive algorithm, each ticket is given a lifetime. Every time a predicate is chosen by the lottery, the ticket of that predicate becomes “older” where the counter for that ticket is incremented. If the ticket reaches its lifetime threshold, it is removed from its predicate. This ensures that predicates must continually remain selective and complete items sooner than the lifetime of the tickets in order to maintain their number of tickets. In essence, this ticketing system acts as a “sliding window” that focuses on the most recent behavior of the predicates and ignores how well each predicate behaved in the far past.

5 Experimental Methods

For the simplification of our experiments, we limited each test into two predicates to test how the ticketing queue algorithm responds to certain configurations of selectivity and cost. In addition, we used a queue size of 1 to easily keep track of which items are in the queues and to simplify the configuration. In addition to our ticketing queue algorithm, we also implemented a few base-line systems to compare our filter to. First, we had a random algorithm that randomly picked chose a predicate when an item had to be routed to a predicate. We chose to implement this algorithm to observe how dynamically filtering each item compared to just randomly picking either predicate. In addition, because each test had two predicates, we also implemented the two possible static filters: one that always chose the “optimal” predicate that was either more selective or cheaper, and the “worst” predicate that had a low selectivity or was very expensive. We used these two filters to gauge how our algorithm did between the best and worst case scenarios.

In these experiments, we chose 5 votes to be the minimum and 21 votes to be the maximum amount of tasks to reach a consensus on a particular item/predicate pair. We chose the uncertainty level to be 0.2. This means that level of uncertainty calculated by the cumulative beta distribution function must be below 0.2 in order for the votes to reach a consensus. Each algorithm was simulated 50 times for each configuration regarding hotel questions and 200 times for restaurant questions since there were fewer items in the list of restaurants.

6 Testing and Results

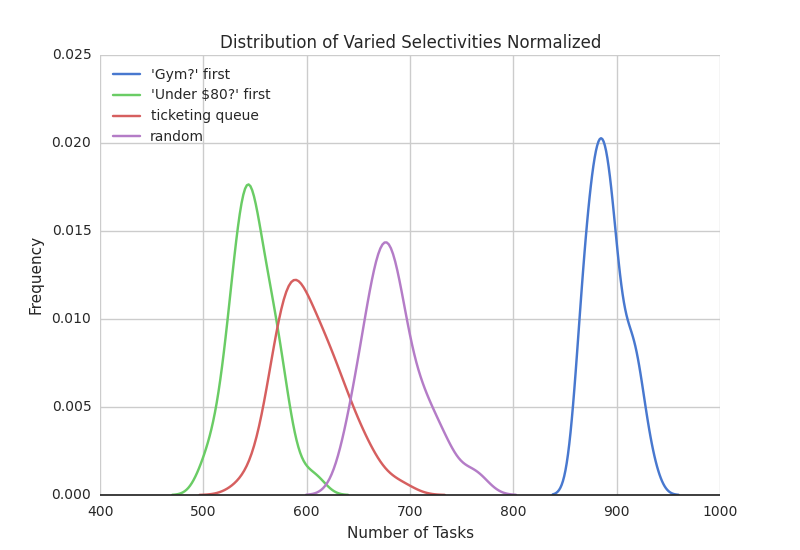
The following section focuses on specific configurations we tested our algorithm with and how our algorithm responded to these configurations. In these runs, we focused specifically on comparing our algorithm to the optimal filter and the random filter. We wanted to determine how much more tasks our algorithm did in comparison to the optimal case, and if our algorithm did significantly better than randomly picking either predicate.

**6.1 Varying Selectivity**

The first configuration we chose was to vary selectivity while keeping the cost constant. This configuration was chosen to observe how the ticketing system of the algorithm affected the number of tasks used by the filter. Both of the predicates chosen had low cost, with an average of 5 votes per item/predicate pair to ensure that the selectivity would be the main contributor to variation in the number of tasks.

|  |  |  |
| --- | --- | --- |
| Predicate | Selectivity | Cost |
| Does this hotel have a gym? | 16% | 5.0 votes |
| Does this hotel cost under $80 a night? | 88% | 5.0 votes |

*Table 6.1: These are the predicates chosen for this configuration. As shown, the average cost is equivalent, but the selectivities are varied.*



*Figure 6.1: This graph shows the distributions of the number of tasks for the various algorithms for varied selectivity.*

Figure 6.1 shows the distribution of the number of tasks after running 50 simulations for the ticketing queue algorithm and the other base-line algorithms. We can see that the ticketing algorithm outperforms the worst-case static filter and the random algorithm, and remains close to the optimal filter where the more selective predicate is always chosen first.

When comparing the ticketing queue algorithm to the random algorithm, we found that the mean number of tasks were 603.84 and 685.52 respectively. When running statistical analysis on the two distributions, we found that the two algorithms were statistically significant.[[1]](#footnote-1)

As shown by the simulation runs, the more often the algorithm chose the “Gym?” question, the less tasks the filter had to assign. The ticketing algorithm averaged around 55 more tasks than the choosing the optimal predicate from the start. Theoretically, this would be around the number of tasks it took the algorithm to learn which predicate was better to choose.

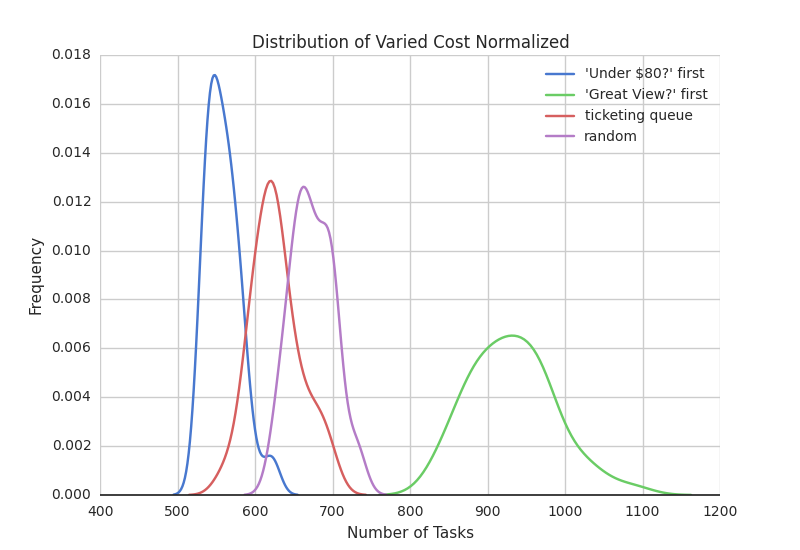
**6.2 Varying Cost**

We also chose to test the algorithm with predicates of different costs and high selectivity. This configuration was used to observe how the queue system of the algorithm, in combination with the ticketing system, responded to the different cost between predicates and if it was able to effectively decrease the number of tasks required to filter out the items. In this case, predicates with higher selectivities were chosen to promote filtering an item out once the item/predicate pair was complete. Otherwise, if the selectivities of both predicates were low, many items would have to be passed to both predicates, regardless of which predicate was chosen first and the algorithm would not be needed.

|  |  |  |
| --- | --- | --- |
| Predicate | Selectivity | Cost |
| Does this hotel cost under $80 a night? | 88% | 5.0 votes |
| Are there great views from the rooms of this hotel? | 62% | 9.7 votes |

*Table 6.2: These are the predicates chosen for this configuration. As shown, the average cost is varied, and both predicates are relatively selective.*

The goal of this configuration was to observe how the algorithm responded to predicates of different costs. The ticketing and queue system worked in tandem to create a backpressure on the more expensive predicate. As the “Great Views?” question required more votes on average to reach consensus on a task, the queue would prevent other items from entering this predicate’s queue until the predicate was finished. Because of the decrease of items routed to the expensive predicate, the other predicate received more new items, which incremented the amount of tickets it received. Thus, these two systems worked together to sense which predicate had a greater cost.



*Figure 6.2: This graph shows the distributions of the number of tasks for the various algorithms for varied costs.*

From Figure 6.2 and Table 6.3, we can see again that the ticketing queue algorithm takes fewer tasks on average to filter than the random and worst case filters. In comparison with the optimal choice, the algorithm only takes roughly 12% more tasks than the optimal filter.

We ran a significance test between the number of tasks of the ticketing queue algorithm and the random algorithm. We found that between the average number of tasks of 626.90 for the ticketing queue and 674.08 for the random algorithm, these two systems were statistically significant.[[2]](#footnote-2) It is important to note that the difference between these two algorithms is less in this configuration than the configuration in section 6.1 because our algorithm is more aggressive when it comes to differentiating between selectivity. In addition, the difference in the costs between the two predicates in this configuration does not range as greatly as the two predicates in the previous configuration. While these two predicates lean towards a higher selectivity, they are not equivalent, which may also affect the differences between the ticketing queue and random algorithm. All these factors taken into consideration, the ticking queue algorithm still significantly outperforms randomly choosing either predicate.

We can see that in this configuration, where the cost is varied between the two predicates, the back-pressure caused by the queue in the ticketing queue algorithm has an effect of routing more items to the “Under $80?” question, or the cheaper predicate. The variance of the worst-case filter is much larger than the rest because an expensive predicate still has a chance to reach consensus early (around 5 votes), but can also require multiple recounts. As a result, constantly choosing this expensive predicate will result in a wider variance in the number of tasks between simulation runs.

|  |  |
| --- | --- |
| Hotel Questions with Varying Selectivity and Low Cost (Gym?, Under $80?) | |
| Ticketing Queue Algorithm | x1.10 |
| Random Algorithm | x1.25 |
| Worst Case Algorithm | x1.63 |
| Hotel Questions with High Selectivity and Varying Cost (Under $80, Great Views?) | |
| Ticketing Queue Algorithm | x1.12 |
| Random Algorithm | x1.21 |
| Worst Case Algorithm | x1.67 |

*Table 6.3: This table shows the multiplier of each algorithm in comparison to the optimal algorithm, calculated by dividing the average number of tasks of each algorithm by the average number of tasks for optimal filter.*

**6.3 Low Selectivity**

We also ran some configurations where the order of the questions had minimal effect on the number of tasks per simulation. In this configuration, the two questions had varying costs and low selectivity.

|  |  |  |
| --- | --- | --- |
| Predicate | Selectivity | Cost |
| Does this restaurant have drinks for those under 21? | 16% | 8.6 votes |
| Does this restaurant have more than 20 items on its menu? | 16% | 5.5 votes |

*Table 6.4: These are the predicates chosen for this configuration. As shown, the selectivities are both relatively low and the costs are varied.*

Because the set of restaurants had fewer items (20 restaurants), the number of tasks required for the filter operation was also less. We decided to run 200 simulations for each algorithm to obtain a more accurate analysis of the data. When running the simulations, the number of tasks outputted by each algorithm were close in value. In further analysis, we found that the average number of tasks between all four algorithms were not statistically significant.[[3]](#footnote-3)

|  |  |
| --- | --- |
| Restaurant Questions with Low selectivity and Varying Cost (Drinks?, Menu?) | |
| Ticketing Queue Algorithm | x1.01 |
| Random Algorithm | x1.02 |
| Worst Case Algorithm | x1.03 |

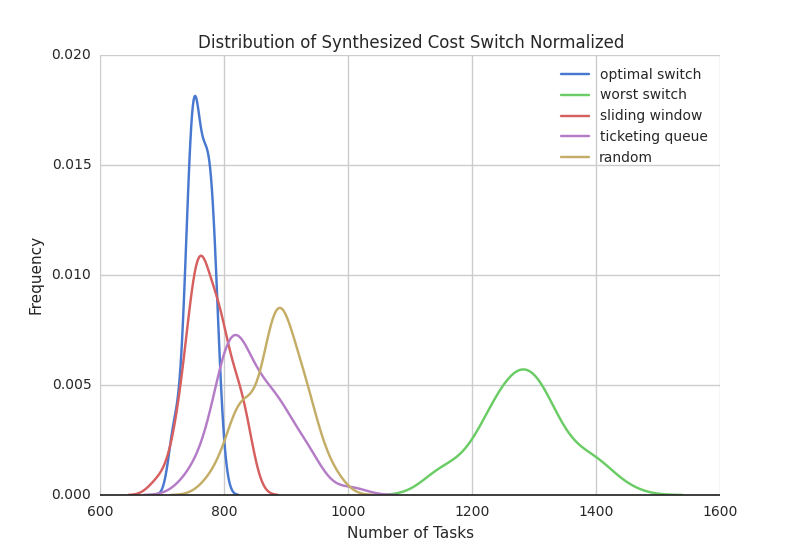
*Table 6.5: A table of the multiplier of each algorithm in comparison to the optimal filter.*

As shown in Table 6.5, the multiplier differences from the optimal filter between the three algorithms are both minimal and are within x0.01 of each other. While the multiplier differences still follow the same trend of the ticketing algorithm outperforming both the random and worst case filters, the difference between these algorithms is not enough to make a significant claim. This is because of the low selectivity of both predicates. Because both predicates have low selectivities, they often return true for items passed to them by the filter. Thus, no matter which predicate the item is routed to first, the item will often need to be routed to the other predicate. As a result, the number of tasks that can be avoided through the ordering of the predicates is minimized due to the low selectivity of both the predicates.

**6.4 Fluctuations and the Sliding Window**

In addition to testing our ticketing queue algorithm, we also wanted to test how our “shifting window” mentioned in Section 4.3 adapted to fluctuations in the properties of the predicate. To test this algorithm, we generated synthetic data using a random generator. Each vote was cast based on the selectivity of the predicate (whether the predicate would lean towards true or false) and added noise that controlled cost. This noise factor was simply the probability of choosing the Boolean that the predicate leans towards. For example, a noise level of 0.5 would be a very expensive predicate while a predicate of 1.0 would be the cheapest possible predicate.

In the synthetic data, there were 2 predicates that were filtering 100 items. Predicates P0 and P1 both had selectivity of 0.9 and a noise level of 0.9 and 0.6 respectively. The predicates switched noise level when the simulation completed 200 tasks. Ideally, an adaptive algorithm would first favor P0, and adaptively choose P1 after the switch. We ran five algorithms - the sliding window, our previous ticketing queue, the random filter, and a best and worst case scenario. In this configuration, we chose to make the switch based of cost not selectivity to simulate a live version as closely as possible. Complex predicates assigned to the crowd are more likely to fluctuate over time in cost than selectivity.



*Figure 6.3: This graph shows the distributions of the number of tasks for the various algorithms for a cost switch at 200 tasks into the simulation.*

This cost switch configuration was tested on 5 different algorithms. First, it was tested on the new adaptive “sliding window” algorithm and the old ticketing queue algorithm. In addition, a few baseline algorithms were implemented to determine how well the developed algorithms were performing. The optimal switch filter chose Q0 first and switched to Q1 at 200 tasks. This way, the filter always chose the cheaper predicate. The worst switch was the opposite, so that the filter always chose the more expensive predicate. The random algorithm was also implemented to test if our implemented algorithms were significantly better than simply randomly choosing either predicate.

From Figure 6.3 and Table 6.6, we can see that the sliding window algorithm outperforms our previous algorithms, and has a 1:1 ratio with the optimal switch algorithm. We also want to check between the old ticketing queue algorithm and random algorithm to ensure that the old ticketing queue system still decreases more tasks than randomly choosing either predicate. While the average number of tasks for the ticketing queue and random queue algorithms were 846.84 and 883.46 respectively. After running a t-test on the two algorithms, we did find that the mean number of tasks between the ticketing queue and random algorithm is significantly different.[[4]](#footnote-4) Thus, our old algorithm can still recover from the cost switch at 200 tasks to outperform the random algorithm, but not as efficiently as the sliding window algorithm.

|  |  |
| --- | --- |
| Synthetic Cost Switch at 200 tasks | |
| Sliding Window Algorithm | x1.02 |
| Ticketing Queue Algorithm | x1.11 |
| Random Algorithm | x1.16 |
| Worst Case Algorithm | x1.69 |

*Table 6.5: A table of the multiplier of each algorithm in comparison to the optimal switch filter.*

Implementing the sliding window functionality may be advantageous when a cost switch occurs in the configuration. However, we also wanted to confirm that adding a sliding window would not be detrimental to the filtering process when there is no switch. We ran the previous tests on the sliding window algorithm and determined the ratio between the average required tasks of the sliding window algorithm with the optimal filter. We then compared the sliding window algorithm to the ticketing queue algorithm to determine if there was a significant difference between the two algorithms.

As shown in Table 6.7, the multipliers of both algorithms are relatively close, which supports the case of using the sliding window algorithm for all cases because it significantly outperforms the ticketing queue algorithm when there is a cost switch in the configuration. Interestingly, the multiplier for the sliding window is lower than the ticketing queue for the varying cost configuration. This may be because the rate at which tickets expire may be greater than the rate at which new items enter the queue for the more expensive predicate. At the very least, the expiring tickets increase the sensitivity for expensive predicates and favors the cheaper predicate more. Thus, the sliding window not only acts as an adaptive method for fluctuations in the configuration, but may also be an added proxy for the differentiation of cost between predicates

|  |  |
| --- | --- |
| Hotel Questions with Varying Selectivity and Low Cost (Gym?, Under $80?) | |
| Sliding Window Algorithm | x1.11 |
| Ticketing Queue Algorithm | x1.10 |
| Hotel Questions with High Selectivity and Varying Cost (Under $80, Great Views?) | |
| Sliding Window Algorithm | x1.07 |
| Ticketing Queue Algorithm | x1.12 |

*Table 6.7: A table of the multiplier of each algorithm in comparison to the optimal filter for configurations of varied selectivity and varied costs.*

7 Discussion and Next Steps

From the multiple tests, we can see that this adaptive dynamic filter effectively decreases the average number of tasks needed perform the filter operation. In every case, our adaptive algorithm performed better than randomly picking either predicate. The ticketing system is able to determine which predicate is more selective and the queue system prevents expensive predicates to take in too many items. The algorithms consistency over a number of different tests proves that the algorithm is not over-fitting to a specific configuration but is truly dynamic. With the sliding window, we can see that the added functionality adds an adaptive component to the algorithm that decreases the average number of tasks when there is a cost switch in the predicates without significantly affecting the algorithm when there are no fluctuations in the properties of the predicates.

The future goal of this project is to implement a live version of the filter that dynamically assigns tasks to workers directly on AMT. There are a few issues with moving this algorithm to a live version. First, there are concurrency issues where two workers may return a task on the same item/predicate pair at the same time. In addition, there are many open-ended questions regarding the completion time of each task and how completion time may also factor into the cost of a predicate. Finally, the live version will have to deal with ensuring workers that they will always receive a task while the test is running. Currently, in the simulation, if a worker has completed all the tasks in the queues, no task is assigned to the worker since there are no tasks currently available. However, workers on AMT must be given a task while the filter is still running, so we must figure how to assign these tasks to workers while maintaining the efficiency of the filter.

References

1. Trushkowsky, B. 2014. Collecting Data With the Crowd [dissertation]. [Berkeley (CA)]: University of California, Berkeley.

2. Sheng, V.; Provost, F.; Ipeirotis, P. 2008. Get Another Label? Improving Data Quality and Data Mining Using Multiple, Noisy Labelers. In *Proceedings of the 14th ACM SIGKDD.* ACM.

3. Avnur, R. and Hellerstein, J. M. 2000. Eddies: Continuously Adaptive Query Processing. In *Proceedings of SIGMOD*. ACM.

4.Auer, P.; Cesa-Bianchi, N.; Freund, Y.; and Schapire, R.E. 1995. Gambling in a Rigged Casino: The Adversarial Multi-armed Bandit Problem. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science.* IEEE Computer Society.

1. The standard deviations of the ticketing queue and random algorithm were 31.37 and 29.06 respectively. The t-value was 13.50 and the p-value was < 0.0001. [↑](#footnote-ref-1)
2. The standard deviations of the ticketing queue and random algorithm were 31.67 and 26.87 respectively. The t-value was 8.03 and the p-value was < 0.0001. [↑](#footnote-ref-2)
3. The following t-values are listed for the ticket queue algorithm vs. the optimal case, random algorithm, and worst case in order: 0.8504, 1.5140, and 1.6902. The p-values are 0.3956, 0.1308, and 0.0918 respectively. [↑](#footnote-ref-3)
4. The standard deviations of the ticketing queue and random algorithm were 54.55 and 45.97 respectively. The t-value was 3.63 and the p-value was < 0.0005. [↑](#footnote-ref-4)