LEARNING-AUGMENTED FREQUENCY ESTIMATION WITH AN ONLINE ORACLE *

Kavya Ravichandran
Department of Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
rkavya@mit.edu

ABSTRACT

Estimating frequency of elements using hash-based methods incurs estimation error when very frequent, or heavy, elements collide with infrequent, or light, ones. Prior work has shown promise in training a learned oracle that predicts the weight of an element and stores the raw count if it is heavy (Hsu et al.). In this paper, we seek to develop such an oracle that is trained online, as stream elements are received. During this work, we found some promising trends in how to batch stream data and how to train accordingly, but results were overwhelmingly marked by high variation. We posit that that is due to significant stochasticity in our system and propose ways to reduce that randomness.

1 INTRODUCTION

In a stream of data, we might want to know how often a given element occurs and perhaps classify whether or not it is a "heavy hitter," i.e., an item that appears very often. A naive solution is to maintain a counter for each element that we see, updating the counter each time we see an already-seen element and adding the element to the dictionary if we haven't seen it before. However, if the vocabulary is very large, we may require too much storage space to build such a dictionary.

1.1 COUNTMIN

The CountMin data structure (Cormode & Muthukrishnan) attempts to solve this problem by hashing every element seen and storing counts associated with the hashes. With well-selected hash functions, the probability of collision can be bounded, and, moreover, using r different hash functions and then taking the minimum count allows for further reducing error probability. However, a major issue arises when a heavy and a light element collide. The estimated count for the heavy element is only offset by a small amount, since its collidee is light, but that light element's count is significantly offset.

1.2 LEARNING-AUGMENTED FREQUENCY ESTIMATION

Hsu et al. showed that we can use a learned oracle to determine whether or not an element was a heavy hitter to improve the performance of a `CountMin` data structure. This heavy hitter oracle was trained offline on a fixed prefix of the data. They showed improvements in accuracy of estimation of frequency within the CountMin due to no collisions with heavy elements. They also showed that the space accuracy trade-off is much better in this setup.

1.3 MOTIVATION AND APPROACH

One limitation of the above approach is that we are not able to fully retain the streaming model; we do require some offline training of the heavy hitter oracle. For use in novel settings, we might benefit from being able to fully maintain the streaming assumption. That is, we want to be able to

*Code available at `https://github.com/kavyar314/onlinecm`

train our model "on the fly."

For this, we might consider using an online machine learning algorithm. However, the unique challenge of this streaming setting is that we do not have labels for our data *a priori*. Upon seeing an element, we do not know whether or not it is a heavy hitter without any context.

We can use the ℓ_1 Heavy Hitter problem as a case study of the general paradigm of using online machine learning algorithms to create data-informed instances of classical streaming algorithms.

2 Design of Solution

In this section, we discuss the core issue of our problem and the general approach we used (Section 2.1). Then, we discuss in detail the data structure we developed for storing training data (Section 2.2). Following this, we describe two different methods we used for training and associated metrics (Sections 2.3, 2.4).

2.1 Core Issue and Approaches for Resolution

At first the setup of the problem seems to easily lend itself to one of many online machine learning algorithms such as perceptron. However, all these online learning algorithms rely on having labels for the data. In this setting, we cannot know whether an element appears many times until it appears that many times. Thus, the issue we face is how to extract labels from the stream. The basic strategy for overcoming this that we use is to consider stream elements in batches. While this is not *truly* a streaming algorithm, we can still do it with sublinear space, and there is precedent for algorithms in the streaming model storing some amount of the stream before taking an action (Guha et al.).

2.2 Data Structure to Store Training Data

The success of this algorithm relies on the quality of the training data and its ability to represent the stream data that we would have gotten in the offline setting. To this end, our major contribution is that we design and implement a data structure to select and store stream elements as they appear. This data structure has three main interesting methods: `add_element`, `sample_elements`, and `check_hh`. In addition, it has the functionality to `decay_n_heavy_hitters`and `flush`. We discuss these in detail and justify our design choices in the section. First, however, we provide a brief overview of the use of the data structure.

During the training phase, every element received is added to and instance of the data structure, according to its constraints. After a batch of elements is received, elements are sampled uniformly such that there are n heavy hitters in the sample and αn light elements in the sample, where $\alpha \in \{1, 2, 3, 4, 5\}$ and elements and their labels are returned as training data. A neural network is then trained on these data. If the loss has decreased sufficiently, then the number of elements stored is decayed and if not, it remains at the previous amount.

2.2.1 add_element

This method adds elements to the data structure and is expected to be called on a stream element. It roughly follows reservoir sampling to ensure the number of stored elements does not become too big. If the element is one of the first k elements to appear, then it is added with a count of 1. If not, then:

- with probability $\frac{k}{s}$, where k is the number of stored elements and s is the number of elements seen so far, add the element to the table and remove a randomly sampled light element.
- otherwise, the new element is ignored.

By doing this, we are randomly selecting light elements to keep, as there may be very many of them, and therefore limiting the amount of storage we require for the training data.

2.2.2 SAMPLE_ELEMENTS

This method is called when training data is required. It takes as an argument whether heavy elements or light elements are desired, randomly samples from the appropriate list, and returns the elements with their labels. Over the course of experimentation, we tried using three different kinds of labels: (1) binary labels referring to whether or not an element was one of the k most frequent ones; (2) frequency as the ratio of number of appearances divided by the length of the stream; (3) the $\log$ of the frequency as defined in (2). In the results section, we mention the performance of each.

2.2.3 CHECK_HH

This method was surprisingly non-trivial to implement. We needed to check first whether there were at least k elements, where k is the number of heavy hitters we wanted to store. If not, then all the elements were considered heavy hitters, and if there were, we then had to decide whether the elements would have been heavy hitters before the decay or not. Indeed, it seems that carefully picking the definitions here for heavy hitters would drastically affect the performance of the system. In Section 5, we talk about how we might differently define heavy hitters or choose training samples based on more granular information about frequencies than just rank.

2.3 CLASSIFICATION

One approach was to simply deem an element a heavy-hitter if its frequency was among the top k-highest frequencies. We varied k to see how this affected training.

2.4 REGRESSION

Another approach involved considering the counts of the elements and dividing by the length of the stream to get frequencies that were then predicted by the model. We considered this both directly and after taking the $\log$ to make differences more stark. There generally were order-of-magnitude differences between the frequencies of the heavy and the light elements, which suggested that the $\log$ frequency was a reasonable label. However, we found that this was difficult to learn, at least with the set of parameter combinations we tried.

3 TRAINING DETAILS

We discuss the various parameters we tested to understand if the approaches were promising (Section 3.1). In Section 3.2, we mention the datasets we used and in Section 3.3, we describe the models we used. Evaluation metrics are presented in Section 3.4.

3.1 PARAMETERS

It seemed evident that this system would not perform well without carefully chosen parameters. For example, if we consider too few elements as a batch of stream elements, then we likely will not see *any* element often enough to understand something about its underlying probability. Thus, we varied the following parameters in the following ways:

- number of samples taken from data structure (which also affected how many samples we stored in the data structure) $\in \{20, 40, 50, 60\}$
- number of samples considered a "batch" of the stream $\in \{50, 75, 100, 200, 500, 1000\}$
- decay parameter that controls how the number of stored elements is decayed according to Equation 1 when loss of the model decreases $\in \{1, 0.99, 0.9, 0.75\}$ (note, this is the fraction by which k is multiplied, so 1 means that there is no decay).
- the number of forward and backward passes through the set of data gathered in a batch $\in \{2, 4, 10, 20\}$

$$\text{n\_elements\_stored} \leftarrow \text{decay\_param} \cdot \text{n\_elements\_stored} \tag{1}$$

Further, we also considered different absolute numbers of elements (initial testing on small numbers of elements and then tried large sets). We also defaulted to resetting the data structure between batches but also tried doing without.

3.2 DATASETS

We tested our approaches on two main datasets. For the classification dataset, we tested on a dataset consisting of the text from the first part (about 3000 words) of *Pride and Prejudice* by Jane Austen (Austen). We simply considered each word to be a stream element. We surmised this dataset would be simpler than other datasets typically used for streaming problems. If we take 1000 consecutive words in a book, then we are likely to see most of the k most common words in English, and these are pretty static. That is, words like "the," "be," and "and" will appear frequently in the prefix and in the rest of the dataset.

For the regression formulations, we used one day of AOL search query data (hosted by Dudek), which consists of over 3 million search queries. We felt this would be more rich in terms of understand search patters. We also began to test the classification formulation on this dataset but were not able to complete prior to the project deadline and therefore we leave that to future work.

The characteristics of the datsets are likely to be quite different. In search queries, we are likely to see some amount of "burstiness." People might search a term, and, not finding a satisfactory answer, might search it or very similar things again, or following a significant event, many people might be searching similar things around the same time. On the other hand, in language, it is very unlikely to see the same words even twice, let alone several times in a row. Thus, we acknowledge that the comparison is not easy. However, for practical purposes relating to the speed of software development using a smaller, simpler dataset, we did benchmark using both. A more thorough benchmarking in the near future will include both approaches on both datasets.

3.3 MODEL

We used three different models all in Keras, one for classification and two for regression. For classification, we used pre-trained GloVe embeddings and therefore just used a fully-connected network of varying depths. For regression we tried both a network with a long short-term memory (LSTM) unit and one with a vanilla recurrent unit [1]. For the fully-connected network and LSTM, we train using SGD, and for the RNN, we use RMSProp.

One subtlety is that when we tried to use frequencies as the label in the RNN, the network produced a guess of 0 for nearly every element. We surmised that this was due to the frequencies being very low to begin with, and so for this model, we trained on the $\log$ of the frequency instead.

3.4 METRICS

For classification, the metrics were easy – we simply count how many test samples are correctly classified.

For regression, devising metrics was more difficult. In the end, due to the preliminary nature of this work, we focused on how well the network was predicting frequencies in an absolute sense, so we considered the difference between the actual and predicted frequencies, thereby defining accuracy as the number that were close to a threshold. For the raw frequencies, we used as thresholds $[0.0001, 0.001, 0.01]$. For $\log$ frequencies, we used as thresholds $[0.1, 0.5, 1]$. The difference in frequencies were an order of magnitude – heavy elements appeared with near 0.001 frequency and light elements with near 0.0001 frequency. Thus, in the $\log$ domain, the difference would be $|\log(0.0001) - \log(0.001)| \approx 2$. Thus, in this domain, a threshold of 1 is still meaningful for discriminating between heavy hitters and light elements.

[1] While both have their merits, the honest reason we used both was because at first, we missed the footnote in Hsu et al. describing their architecture in more detail.

4 RESULTS

In this section, we present results for three configurations: a fully-connected binary classifier with simple data, an LSTM trained on frequencies of AOL query data, and an RNN trained on log frequencies of AOL query data relative to various parameters of the system. In particular, we describe the performance of the oracle on heavy hitters under different parameter configurations and comment on clear trends and lack thereof. We evaluated heavy hitters and light elements separately, since the performance of the composite system relies differently on the accuracy on each of these [2]. In general, trends were stronger on heavy hitters than on light elements, so we present those. In Section 5, we comment on anomalies seen in the performance on light elements.

Overall, we find first, performance is not reliably good, and second, that there are few trends and correlations between the performance and the parameters we considered, and so here we present a sampling of plots that show that. In this section, we discuss heavy hitters; in the Appendix we present similar plots for light elements.

Each point in a given plot corresponds to one unique set of parameter settings. In a given plot, we present the performance as relates to one of those parameters. We make a somewhat naive assumption that the parameters do not interact with each other, and so if we expect to see a trend in one, we can see it regardless of the other parameter settings. Indeed, the variability we see even with repeated experiments of the same parameter setting suggests that the problem is not related to parameter settings but more intrinsic to the design of the system.

4.1 BINARY CLASSIFIER

As mentioned earlier, this dataset is significantly easier than other streaming settings. Here, it would suffice for the network to memorize the frequently-appearing words, and performance would be quite good. Indeed, we see rather good performance for most settings of the parameter for batch size (Figure 1), number of passes (Figure 2), and decay parameter (Figure 3), but none of these is likely to generalize to other, more canonical streams of data.

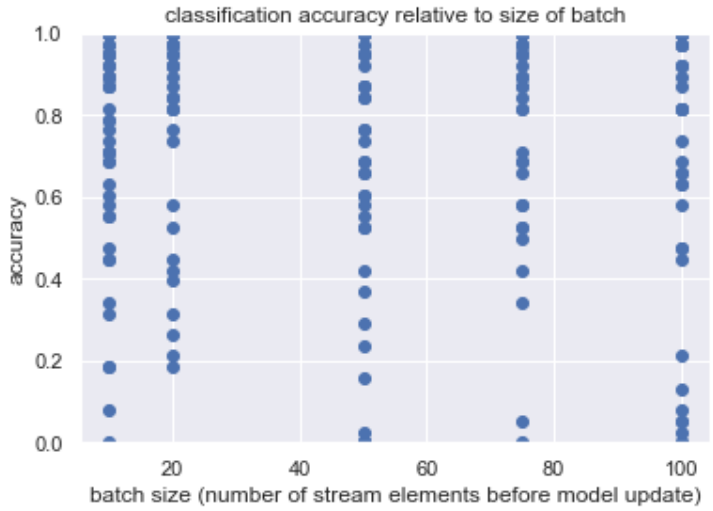

Figure 1: Classification model performs relatively well on *Pride and Prejudice* dataset regardless of batch size.

[2]Improved performance on heavy hitters will reduce the error rate in the CountMin, whereas improved performance on light elements will reduce the storage space required for the lookup table.

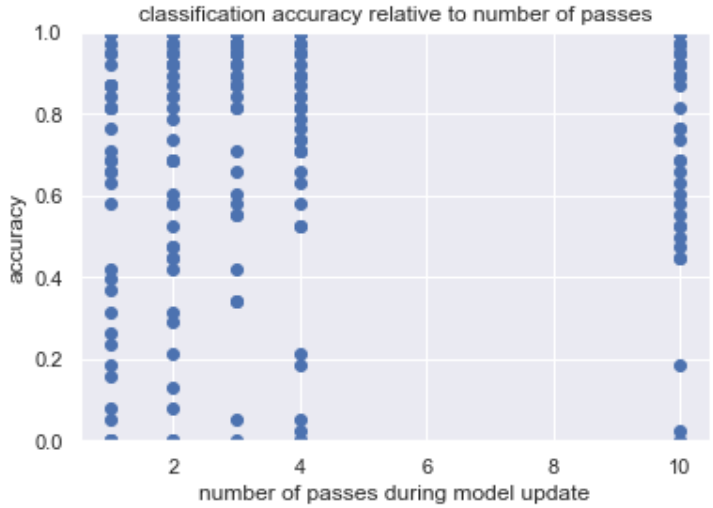

Figure 2: Classification model performs relatively well on *Pride and Prejudice* dataset regardless of number of passes over a dataset.

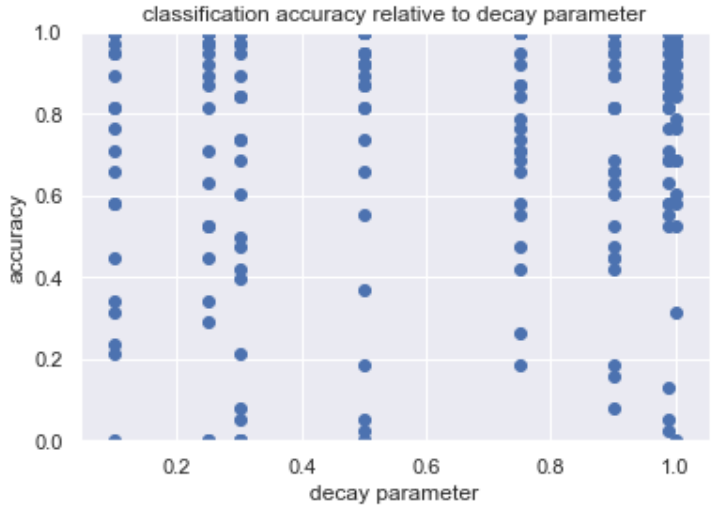

Figure 3: Classification model performs relatively well on *Pride and Prejudice* dataset regardless of decay rate.

4.2 LSTM Frequencies

In this configuration, using an LSTM trained on frequencies of elements from the AOL search query data, we noticed that regardless of other parameters, seeing 200 elements prior to training the neural network afforded better performance on average (27%, compared to 13% for 100 elements and 18% for 500 elements), though the distributions themselves had significant overlap (Figure 4). It appeared that 20 passes through a given set of data performed best on average (Figure 5). Low values of the decay parameter performed more poorly (Figure 6).

4.3 RNN log Frequencies

In this configuration, using an RNN trained on log frequencies of elements from the AOL search query data, we noticed that regardless of other parameters, seeing 200 elements prior to training the

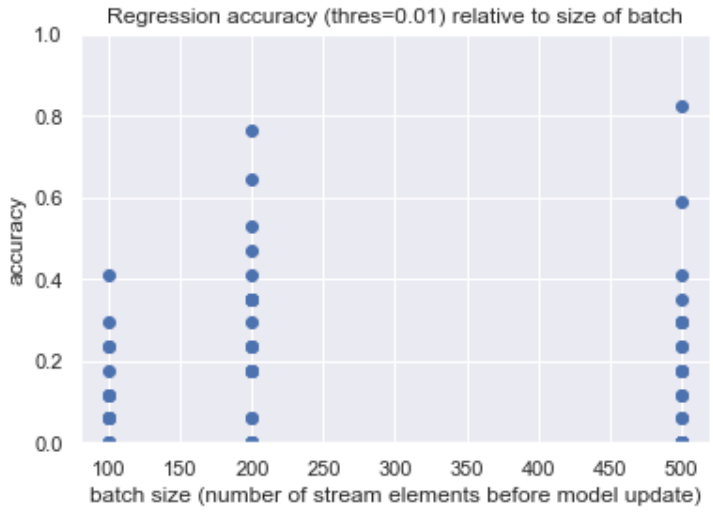

Figure 4: Excepting the outlier in batch size = 500, a batch size of 200 affords better performance than amounts lower or higher on the AOL Search Query Dataset. This is likely because a smaller batch would have noisier estimates of the true frequency while a larger batch would allow for fewer sets of data in a fixed training stream length.

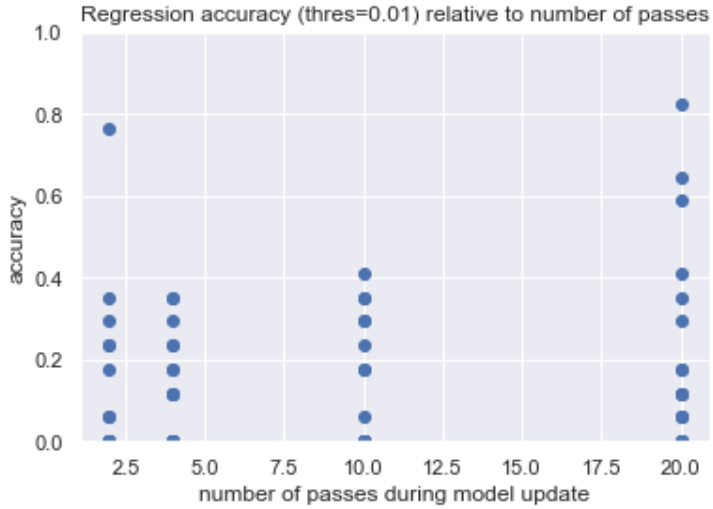

Figure 5: Twenty passes over every set of data performs better than fewer passes over the data on the AOL Search Query Dataset. This makes sense, since we require repeated passes to converge to good weights.

neural network afforded better performance on average (36%, compared to 11% for 100 elements and 26% for 500 elements), even more starkly than in the LSTM case (Figure 7). It appeared that 20 passes through a given set of data performed best by a little (Figure 8). The decay parameter had little effect (Figure 9). There are still significant overlaps in the distributions.

4.4 Summary

Over these experiments, we found that for heavy hitters, a couple of trends held. In terms of the number of elements seen before updating the model, 200 elements tended to be good, at least on average. Performance on heavy hitters tended to improve when increasing the number of passes

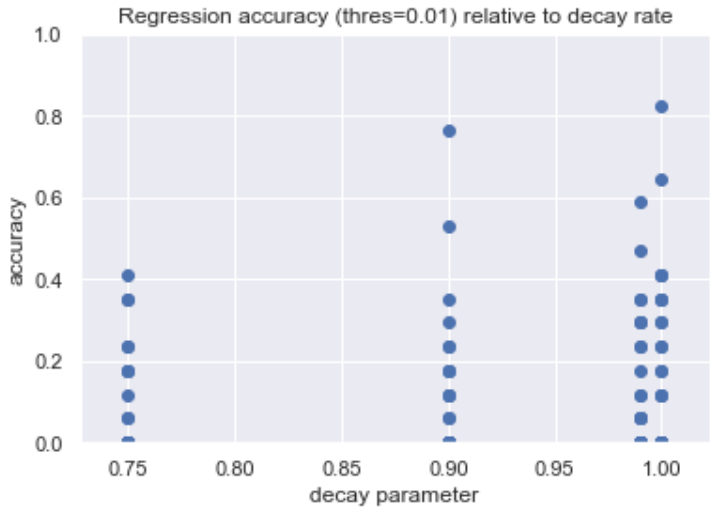

Figure 6: The lowest decay parameter appears to perform less well than higher decay parameters, though variance is still high, on the AOL Search Query Dataset.

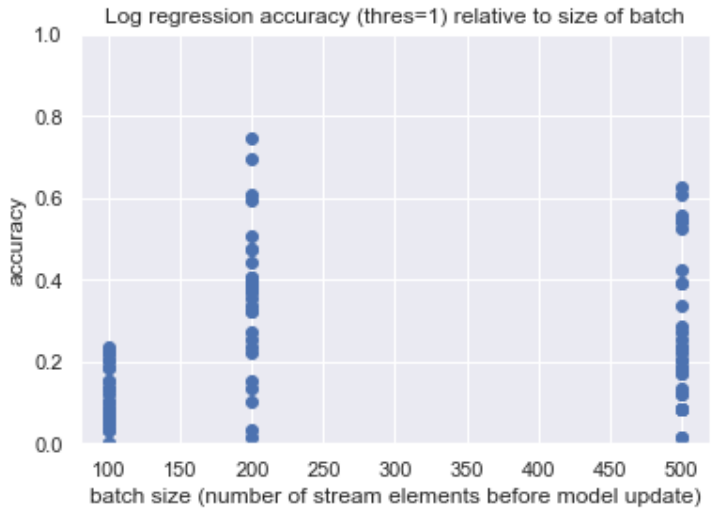

Figure 7: Batch size of 200 performs better than the others on average, more starkly than in the LSTM case, on AOL Search Query Dataset. However, large variance is still seen.

over a single dataset; for light elements (plots provided in Appendix), it was more unclear. Lower values of the decay parameter performed the same or worse than higher ones. However, while these trends hold *on average*, for any given parameter setting, we saw significant variance.

Table 1: Summary of Results

Parameter	**Trend**
Batch Size	200 elements per batch does best, more starkly in RNN than LSTM
Number of Passes	20 updates does best on average, more starkly in LSTM than RNN
Decay Parameter	1 (no information loss) does not differ much in distribution from other values

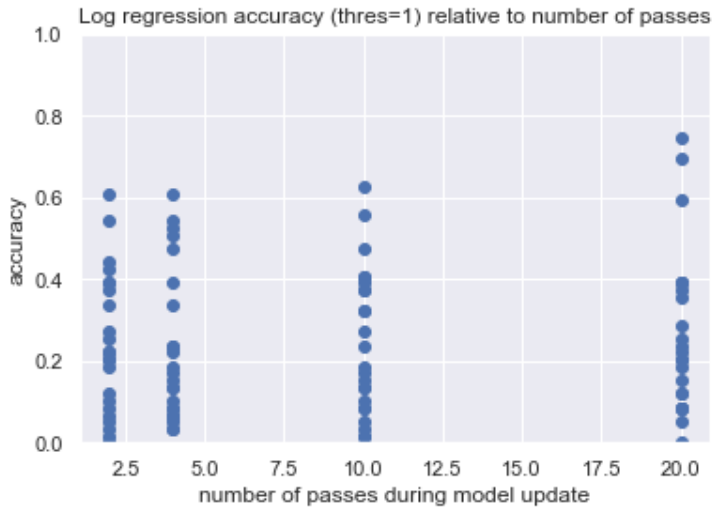

Figure 8: There is no clear trend in how the number of passes affects the performance of the RNN trained on log frequency of the AOL Search Query Dataset.

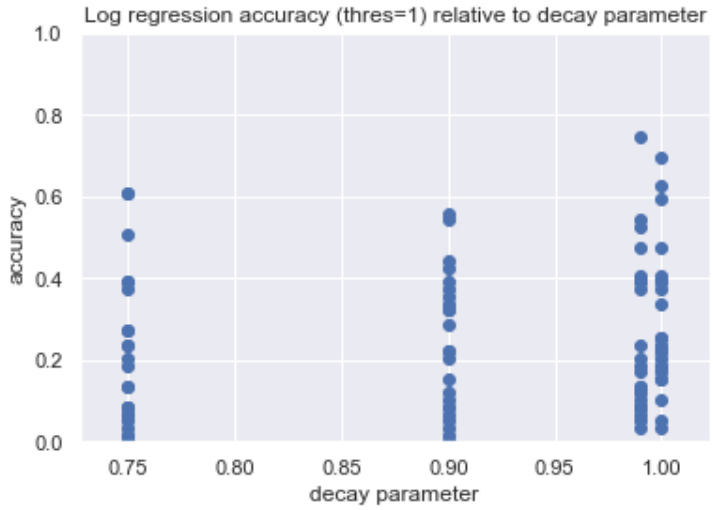

Figure 9: There is no clear trend in how the decay parameter affects the performance of the RNN trained on log frequency of the AOL Search Query Dataset.

In order to understand whether the wild ranges we saw were related to parameter interactions, stemmed from issues with our experimentation, or were inherent to the system, we ran the "optimal" parameter settings 13 times [3]. We noticed enough variation that it confirmed our suspicions that there is underlying stochasticity that parameter settings cannot overcome. We present a plot of accuracy as it relates to batch size in Figure 10, but clearly the same trend would hold along the x-axis regardless of what is plotted, since the parameters were fixed.

[3]There is no magic or superstition to this number. We ran it three times, one-by-one, to see if it was reproducible, and then seeing it was not, ran it ten more times to get an interesting plot.

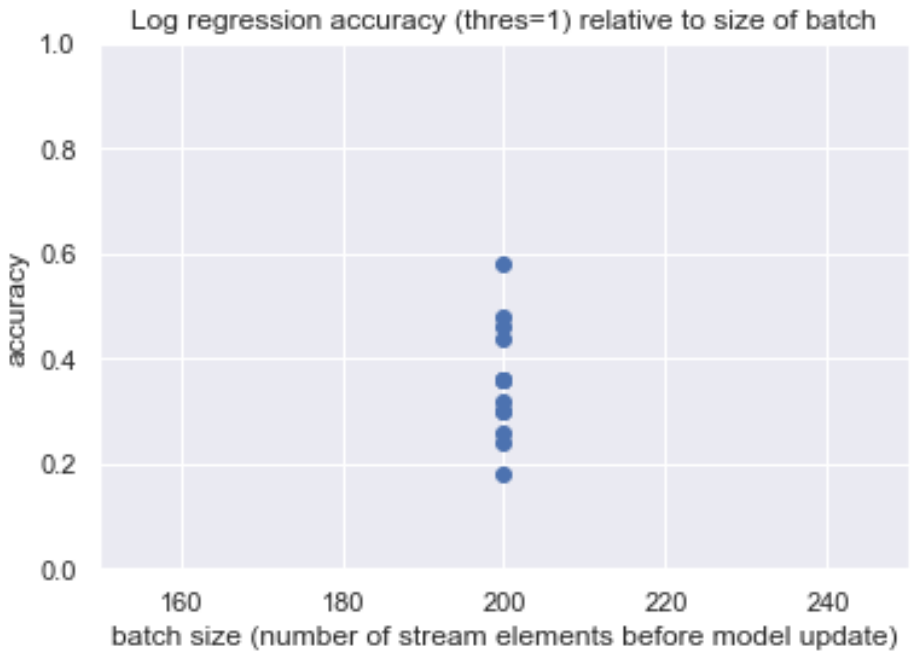

Figure 10: For identical parameter settings, we still see significant variation for different runs, suggesting problems in underlying stochasticity of the system.

5 Discussion and Next Steps

In this section, we discuss the implications of the results, possible explanations for poor performance, and next steps to try to further understand and improve performance.

As mentioned, results were extremely heterogeneous, with some trends in parameters but often not consistently reproducible. Our system also has a great degree of randomness and is susceptible to other sources of randomness. Thus, it seems possible that reducing this randomness and decreasing susceptibility to randomness would help improve performance and make it more reliable. We uncover possible sources of randomness and make recommendations on how to ameliorate them.

The major area in which there is significant randomness entering the system is in data selection. When we first see an element, we only keep it with some probability. Thus, our counts are not perfect. We could also imagine an adversarial setting in which late in the stream, we see the same element repeatedly, but each time, since the probability of not including the element is high, we don't include it. This might cause issues in the quality of light and heavy elements we store in the data structure.

Then, in terms of training we are currently randomly choosing heavy hitters and non-heavy hitters and training on those. However, this means that we might be seeing the heaviest heavy hitters and the lightest light elements, which might reduce the ability of the oracle to discriminate. Thus, it might be worth finding hard examples to train on, which in this case would refer to light heavy hitters and heavy light elements. This would allow us to learn the function well at the boundary. Adding some amount of determinism in these ways might improve the ability to be reproduced of performance under repetition.

Further, it would be interesting to probe the data to better understand what kinds of stationarity assumptions we can make on the process generating the stream. This might help inform how to pick training data.

One other idea would be to instantiate an ensemble of the data structures holding training data to try to exploit the "on average" performance. However, this would be limited by space restrictions.

6 CONCLUSIONS

We sought to develop a learned oracle for frequency estimation that was trained online. While we were able to find settings that performed better than others (but still not objectively well), something underlying about the system causes extensive variation even when parameters are held constant. In future work, we will make aspects of the system more deterministic and understand how performance is affected.

ACKNOWLEDGMENTS

Thank you to Professor Indyk for discussions regarding how to define heavy hitters. Thank you to Professor Daskalakis for discussion regarding how to select training data in the regression case. Thank you also to Chen-Yu Hsu for pointers regarding the dataset.

REFERENCES

Jane Austen. Pride and prejudice. URL https://www.gutenberg.org/files/1342/1342-h/1342-h.htm.

Graham Cormode and S Muthukrishnan. Approximating data with the count-min data structure. pp. 9.

G. Dudek. Index of /~dudek/206/logs/AOL-user-ct-collection. URL http://www.cim.mcgill.ca/~dudek/206/Logs/AOL-user-ct-collection/.

S. Guha, A. Meyerson, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams: theory and practice. 15(3):515–528. ISSN 1041-4347. doi: 10.1109/TKDE.2003.1198387. URL http://ieeexplore.ieee.org/document/1198387/.

Chen-Yu Hsu, Piotr Indyk, Dina Katabi, and Ali Vakilian. Learning-based frequency estimation algorithms. URL https://openreview.net/forum?id=r1lohoCqY7&noteId=Skl8UhSqaX.

APPENDIX

We present the same plots as in Section 4 for light elements.

CLASSIFICATION

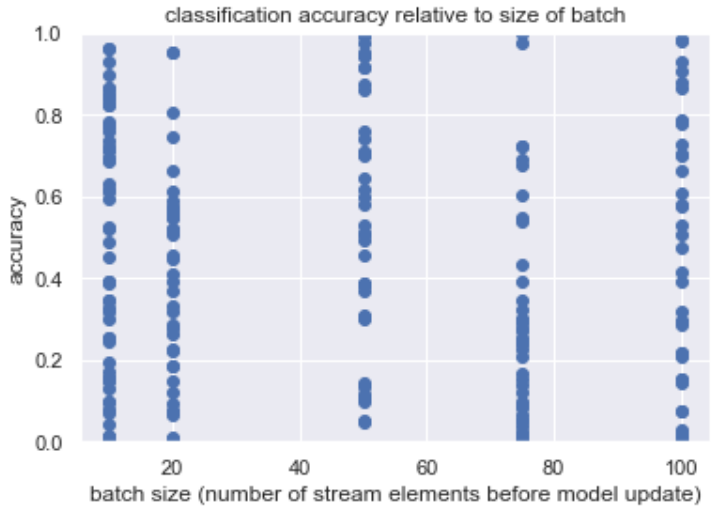

Figure 11: Classification model performs relatively well on *Pride and Prejudice* dataset regardless of batch size.

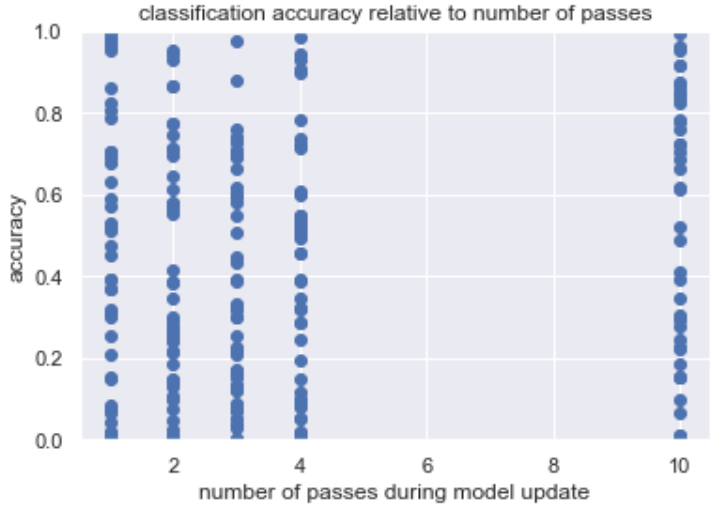

Figure 12: Classification model performs relatively well on *Pride and Prejudice* dataset regardless of number of passes over a dataset.

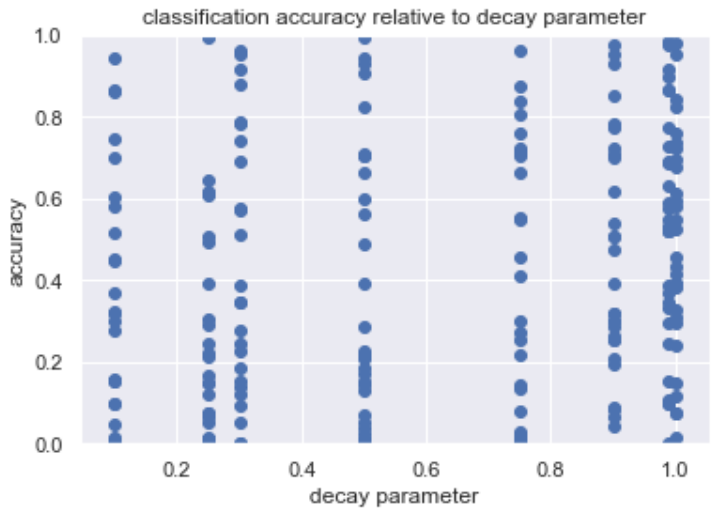

Figure 13: Classification model performs relatively well on *Pride and Prejudice* dataset regardless of decay rate.

LSTM FREQUENCIES

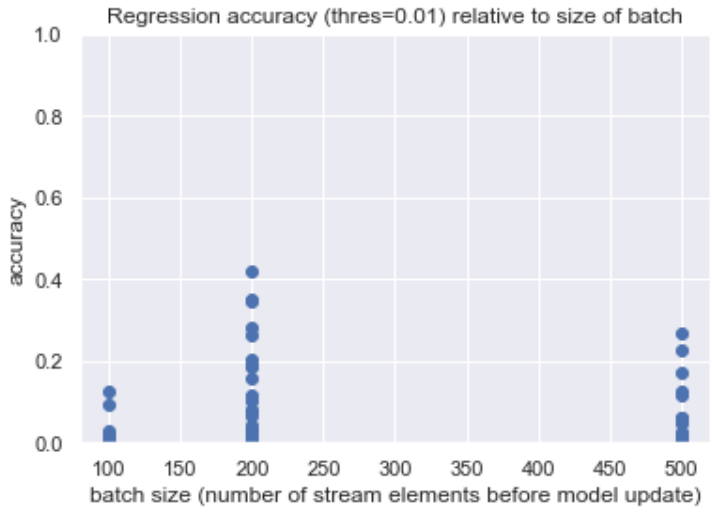

Figure 14: A batch size of 200 affords better performance than amounts lower or higher on the AOL Search Query Dataset. This is likely because a smaller batch would have noisier estimates of the true frequency while a larger batch would allow for fewer sets of data in a fixed training stream length.

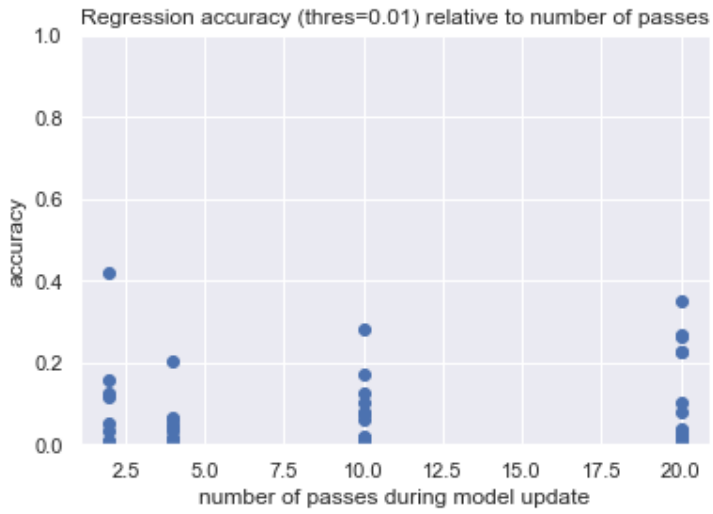

Figure 15: Twenty passes over every set of data performs better than fewer passes over the data on the AOL Search Query Dataset, excepting an outlier when one pass. This makes sense, since we require repeated passes to converge to good weights.

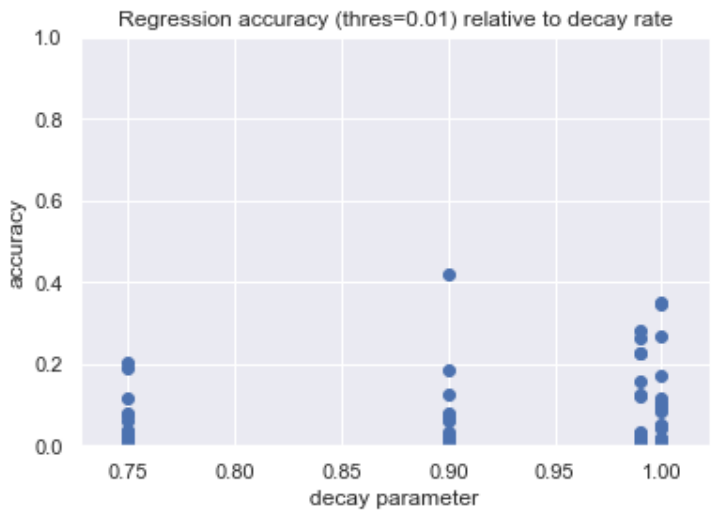

Figure 16: The lowest decay parameter appears to perform less well than higher decay parameters, though variance is still high, on the AOL Search Query Dataset.

RNN log FREQUENCIES

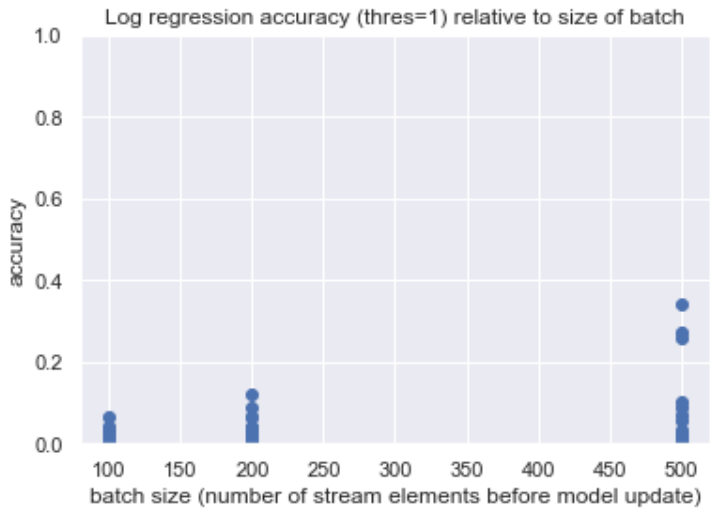

Figure 17: Batch size of 500 appears to perform better than the others on average on AOL Search Query Dataset. However, large variance is still seen.

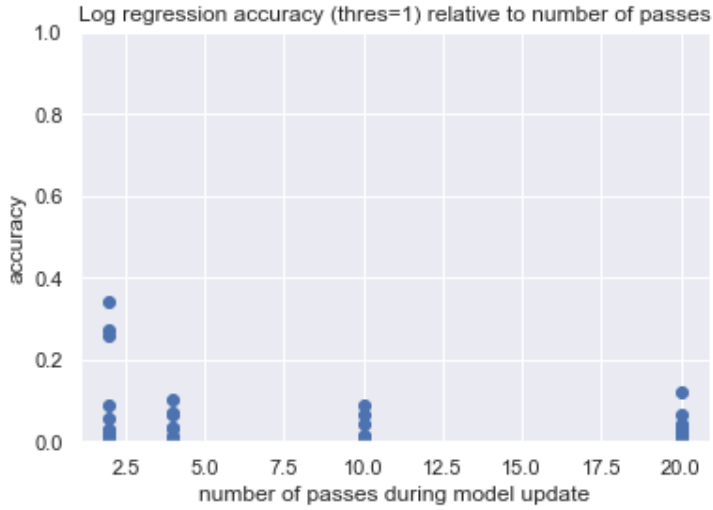

Figure 18: One pass appears to perform best on light elements in the RNN trained on log frequency of the AOL Search Query Dataset. This might be due to the large variety of light elements, and large numbers of passes might result in overfitting.

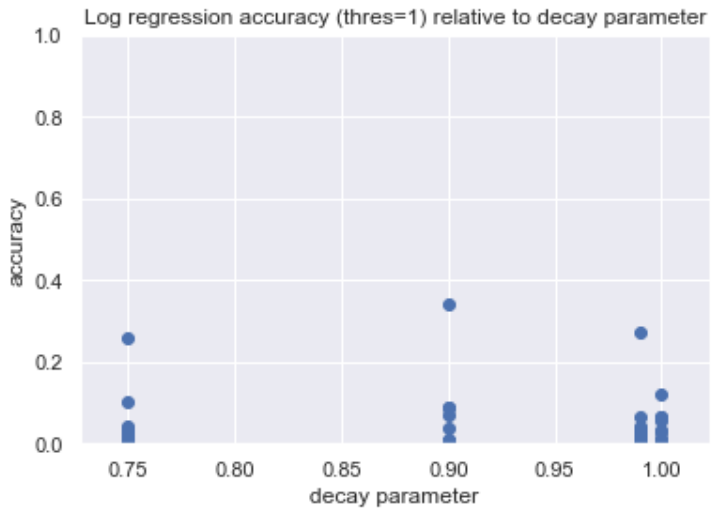

Figure 19: There is no clear trend in how the decay parameter affects the performance of the RNN trained on log frequency of the AOL Search Query Dataset.

SUMMARY

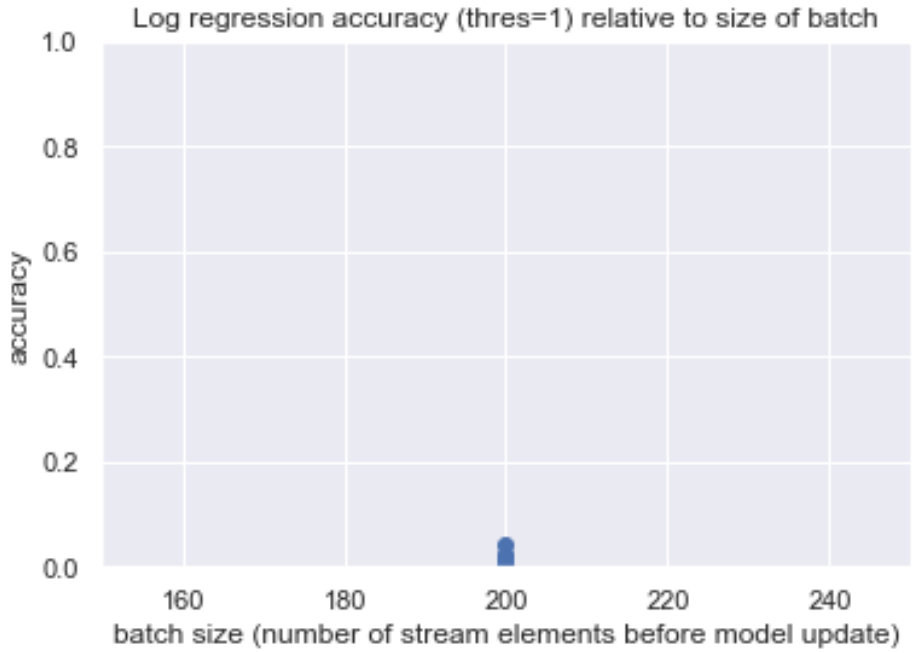

Figure 20: For identical parameter settings, we still see some variation for different runs, suggesting problems in underlying stochasticity of the system.