Luke Phillips
Jacob Ormsby
CS 4964

Final Project Report

Introduction

This project takes inspiration from the paper "The Case For Learned Index Structures" (Beutel, Chi, et al., 2018). In their paper, the authors explore ways to improve the efficiency of various data structures that make use of indexing using machine learning. Our project in particular aims to replace the traditional hash function of a hashmap data structure with a learned index structure. To do this, we implemented a recursive model index (RMI) to learn the cumulative distribution function (CDF) underlying the data distribution of our dataset. We then implemented a hashmap using Python, once using a traditional hash function, and once using the learned index model. We measured the performance of our model by counting the number of conflicts when constructing the hashmap, while also counting the average conflict rate when inserting new values.

Background

Hash Maps are a data structure which maps keys to values, normally using a hash function and an array. The hash function deterministically converts a key of any data type into an index in the underlying array. In each index in the array, a value is stored of any data type. Hash maps have the advantage of having a constant lookup and insertion time complexity. However, they can slow down when multiple keys are mapped to the same index in the array. This is referred to as a collision, and must be handled using separate chaining, rehashing, or linear probing. As such, hash maps perform best when the collisions are kept to a minimum. In practice, an ideal hash function will have a perfectly random and uniform mapping of keys. This results in a predictable collision rate for a given percentage of the array being populated.

For specific data distributions however, a random uniform mapping of keys may not be the most efficient option. If the keys follow a specific probability distribution, for example, it may be possible to make a hash function with a distribution which avoids conflicts more than a uniform distribution could. This is what our approach seeks to achieve. In order to create a learned hash function, we use a recursive model index (RMI) to learn the underlying distribution of our dataset.

The RMI is most accurately understood as a hierarchy of Neural Networks arranged in a tree structure. The head of the tree consists of one neural network, while the subsequent levels can have an arbitrary number of neural networks. Given a key, the first NN will 'map' to a NN in the next level, and so on until the bottom stage of the RMI is reached, where the final NN will map the key to a specific value. Each NN can be of any complexity. This type of model can be used for a range of applications. It is somewhat analogous to a B-Tree, the difference being that the process of mapping to the next node is more complex, and each node can be mapped to by multiple NN in the stage above it. As such, nodes in the model can be replaced with B-Trees once a certain threshold of error is surpassed. This is often done mainly on the final stage of the model. The specifics of our RMI is outlined in the design section.

As for the distribution of the keys, we use an RMI to model the cumulative probability distribution (CDF) function of the data. Given a probability density f(x) the CDF

$$F(x) = \int_{-\infty}^{x} f(x)dx$$

denotes the probability that a number in the distribution is less than x. We map each key using the formula h(K) = F(K)*M where F is the CDF, K is the key, and M is the number of indices in the hash map. If our RMI were able to learn the CDF of the data distribution perfectly, then using it as part of the learned hash index would perfectly map all the keys with no collisions.

Data used

We used the "Enron Email Dataset" from Kaggle for our project. With over 500,000 rows, it has a substantial size that is also not too difficult to work with. It also has only two columns, which bodes well for use in hashmap. It should be stated that the actual data used for this project is not of high importance, as our project seeks to design a data structure that can be used on any 'key value' dataset. No specific information in the dataset actually affects our project, and our only motivation for using it is to demonstrate our technique in a real world scenario.

Motivation

The motivation behind our project is to reduce conflicts generated by the traditional hash function by using a learned hash index model to more evenly distribute the data in the hashmap. Not only will this improve future lookup times by avoiding incurring cache misses, but it will also more effectively use all of the allocated space in the hashmap before needing to grow. This project is interesting as it uses a novel approach with machine learning to replace traditional hash functions with a learned index model using the CDF (cumulative density function) of the data distribution. This project is important with large datasets as any reduction in hash conflicts will result in better space management and faster lookup times. We hypothesize that using a learned index model we will be able to reduce conflicts when inserting new data and speed up lookup times as compared to the traditional hash function method.

Design

Now we will go into what we have accomplished with our design and prototyping. Firstly, the authors of the original paper used an implementation of an industry grade hash map, we on the other hand focused on a basic separate chaining hash map. This hash map implementation was provided from "Hash Map in Python" (Geeks for Geeks, https://www.geeksforgeeks.org/hash-map-in-python/#). Using this basis of a traditional hash map we have performed multiple edits on it to accommodate a constructor that will initialize the hash map with values given an array of keys and values, removed unnecessary code from the class not pertinent to our project, and modified the code basis to allow for collision counting. From there our cleaned dataset (Enron Email Dataset from Kaggle) was loaded into the traditional hash map with the (string) keys being the file column and the (string) values being the message column. We noted that of the 460,937 unique rows, 229,663 of them resulted in a hash collision. As such, about 50% of the keys were involved in a hash collision. In attempts to reduce this number of collisions, we have started implementation of the learned hash map.

The learned hash map decomposes the index model of the data into a CDF model. As such we have defined two other new classes, RMI3 and RMINode. RMI3, standing for "recursive model index – depth of 3" acts as a B-Tree like structure which is composed of RMINodes. Each RMINode handles a subset of the data, processes the CDF of this subset, and passes the desired key down until the leaf nodes at depth 3 are reached. Firstly, the root RMINode at depth 0 will receive a sorted key array. Each RMINode that is not a leaf node will have two children, a left child RMINode and a right child RMINode. The subset of data that each node receives is the left half of its parent dataset if it's the left child or right half of its parent dataset if it's the right child. To calculate which one the key will get passed down to, each RMINode has a logistic regression model to predict the location of the key of interest in its CDF of subset data. If the key is found to have a probability (Pr(X <= key)) <= 50, it will be passed down to the left child, otherwise it will be passed to the right child. Since we are utilizing RMI-3, there are 2^3 = 8 leaf nodes that each contain one of the sorted n/8 size subsets in sequential order. Now at this point the CDF has been narrowed down enough to get an actual position of where the key should go / be found in the backing array of the HashMap. So through utilizing the numpy library, we can use numpy.cumsum(data_subset) to get the equivalent of Pr(X <= key) * M where the probability is the chance of finding the key of interest or anything less than it and M is the size of the data_subset, we then must add that to an offset to account for which of the sorted subsets the key is in. As such, when traversing down the RMI3 tree, a left child movement will append a 0 to a binary number and a right child movement will append a 1. For example, if there are three left child movements, a binary number of 000 will be produced, which means we are in the first subset. So to formalize this, K(i) = Pr(X <= key) * M + offset * subset_size, with K(i) representing the position of the key in the backing array.

Now to apply this to the Enron Email Dataset, since there are 460,937 unique keys we will apply a label encoding to each key to get numbers 0 through 460,936. These integers will form what we believe to be a uniform distribution, as the distance between adjacent integers will always be 1. Then, we can give the encoding of a key to the learned hash map to get its position in the backing array. We also plan to artificially generate a normal distribution set with floats as keys and randomly generated strings as values for an additional test of the learned hash map.

We have found one glaring problem in our current implementation. Random variables require a mapping from the sample space to the range of real values. Our keys from the Enron Email Dataset are in string format, to fix this we plan to label encode them as stated above. However, we must have a way to track these encodings for future key lookups, and our only solution currently is with a dictionary. As can be noted though, using a dictionary (essentially a traditional hash map) within our learned hash map will still result in the same number of collisions for the encoding dictionary. We do realize this is a problem but have yet to find a way to address it. Any feedback with this issue would be greatly appreciated. It is notable however that numerical data as keys will still work fine, such as the float keys with the normal distribution dataset.

Unfortunately, we did not use neural networks. This is because we have no experience with neural networks, we are both complete beginners in machine learning, they are quite complex, we will not be covering them in class, and Professor Rezig suggested that we use the intuition behind the neural networks instead of using them directly.

Evaluation

The first order of business in the evaluation of our implementation was to verify that it worked properly as a hashmap. We verified this by inserting a number of keys and values into the hashmap and testing whether looking up the value returned the same value that was inserted. Once we verified that the hashmap was functioning correctly, we moved on to the tests.

We evaluated the success of our implementation based on the prevalence of conflicts of the learned hash function vs the traditional hash function, both using separate chaining. We also measured the lookup times of looking up each key in the hashmap in random order in both the learned and traditional implementation, then comparing.

In the first test, we constructed a hashmap using the Enron Email Dataset, using the ‘file’ column for the keys and the ‘message’ column for the values. We implemented a counter in our hashmap class in order to keep track of all the conflicts that result from insertion during the construction of our hashmap. We did the same for another hashmap, but instead using a learned hash function instead of the normal random one.

The results were disappointing to say the least. Our learned hash map performed substantially worse than the traditional hash map on construction. Upon creating a hashmap with an RMI depth of 3 and ~500,000 indices, the same as the number of keys we inserted, we had 230729 hash collisions using the traditional hash map, and 806618 collisions using our hash map. This amounts to 0.5 collision/insertion with the traditional hash map, and 1.74 collision/insertion with our learned hash map.

As another experiment, we tested our learned hash map with an RMI depth of 2 and 4. Theoretically, the number of collisions should decrease with an increase of RMI depth as the model has more predictive precision. However, the hashmap with an RMI depth of 2 yielded a collision rate of ~1 collision per insertion, while the hashmap with an RMI depth of 4 yielded a collision rate of 3.25 collision per insertion

We think that these poor results are due to our initially using the 'uniform' CDF are a perfectly linear line, as such more models might actually be leading to more uncertainty which increases hash collisions unnecessarily since there isn't a lot of variation in the CDF. An increased value of 'k' may just be leading to further uncertainty rather than flattening out variation in the CDF (since there is none).

Alternatively we tried to create a hashmap using randomly, synthetically produced data from a normal distribution, as well as using a normal distribution for our CDF in our RMI, and again compared the results to a traditional hash map. In this case, we achieved a collision rate of 0.52 with the regular hash map, and 200 with our learned hash map.

This was actually a huge difference. Although 200 is still small enough to not be considered proportional to the input size of 100,000, considering that the average for the traditional hashmap was 0.5 collisions per key, there is a huge difference in performance. Our hypothesis after running this and doing many debugging tests is that the models in the RMINodes may be having a difficult time with the normal approximation method. There are other methods to try, but we ran into a lot of deadends and eventually just ran out of time.

Finally, using the best learned hashmap with a 'k' of 2 containing the enron email dataset, we compared its runtime to lookup 100,000 keys in random order to the same on the traditional hashmap. We saw a lookup time of 0.1396 seconds with the traditional hash map, and a lookup time of 5.9247 seconds with our hashmap. Analyzing this runtime, it does seem that the traditional hashmap has better data distribution than our learned hashmap does. Although it isn't perfect, this was definitely a great learning experience and has lots of room for improvement in the future with different models and neural networks.

Conclusion

The evaluation of our implementation yielded very disappointing results. The original paper that our work was based off of achieved a substantial *reduction* in hash conflicts, up to 77.5% in one case. In retrospect, the difficulty of this project was definitely above the undergraduate level. Implementation oversights and conceptual errors were probably to blame for the underperformance of our hash map. In future work it might benefit us to take this project and iron out the inefficiencies.

Conceptually speaking, the main strengths of this work lie in applications where conflicts in the underlying hash-map architecture are very expensive. If we were actually able to decrease the number of collisions, then any additional overhead or reduction in performance caused by the machine learning models would be outweighed by the reduction in collisions.

We were able to demonstrate the weaknesses of this method with exceptional clarity. Firstly, we showed that the implementation of this technique, as with any that uses a lot of machine learning, is challenging and often yields suboptimal results. Furthermore, if the data does not follow a well defined distribution, but is more random in nature, the whole premise of the technique breaks down. In this case, any machine learning approach will not be able to index keys with greater efficiency than a regular hash function.

This project was difficult, fun and exciting to work on. It showed us a new and creative application of machine learning that shows we can rethink many commonly used data structures across computer science. Besides actually getting our code to work, future work may include trying a similar technique to improve the performance of B-trees, bloom filters, or other common data structures whose data may be able to be used more intelligently.