**An Explanation for Learned Index Structures**

**Introduction**

This paper is an explanation to the research paper titled, “The Case for Learned Index Structures” by Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis [1]. I will be giving insights on what I found interesting and an explanation of how learned indexes work based on their paper. In the paper it explains about the learned data structures of B-trees, hash maps, and bloom filters using machine learning. I will go in depth about learned B-trees, and learned sorting.

**Background**

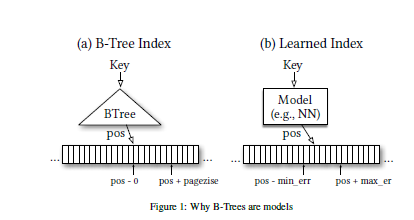
The inspiration behind learned index structures or algorithms comes from the fact that data structures and algorithms do not consider the data they are holding. These traditional data structures and algorithms are great and work well with any type data you throw at it. This is great for versatility. You will also know the worst-case performance. However, it might not be the best for time. What if machine learning can “create its own data structure or algorithm” tailored to given data? Using that data structure might be faster.

For example, say the you want to get the index of a key. To make things simpler the example will be a binary tree. If the data is linear. Let’s say from 1 to 1,000,000. You can build a binary tree for that. However, you could also just return key-1 to get the position of the key, since arrays start at 0. With a binary tree you would have to in the worst case go through the ceiling of log\_2(1,000,000) traverses through the tree which is 20. Binary trees have a time complexity of O(log n). Just returning key-1 is a lot faster than traversing through 20 nodes. That is the basic intuition of using machine learning to enhance the data structure of a B-tree.

**B-tree**

**How Learned B-trees Work**

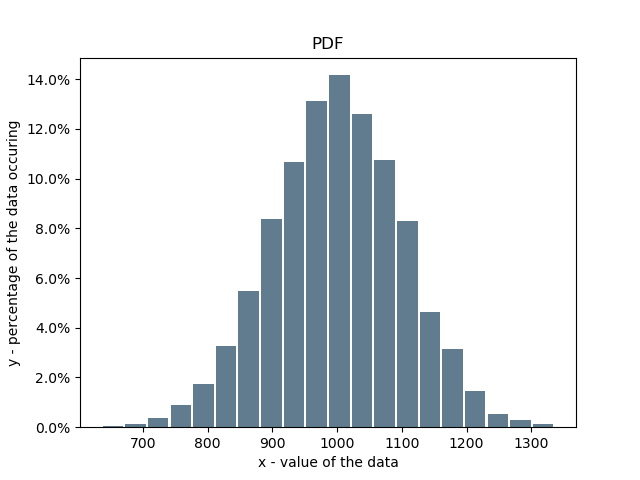
This is how learned B-trees will work. A B-tree is a data structure that given a key to look up it will return the position of that key. Say you want to access 1,000 in the array, it will give you the index of that number. To add a level of abstraction given a key it will predict a position. Machine learning algorithms such as neural networks can all predict a position given a key. Essentially both are models with black boxes in the middle.

Figure 1.

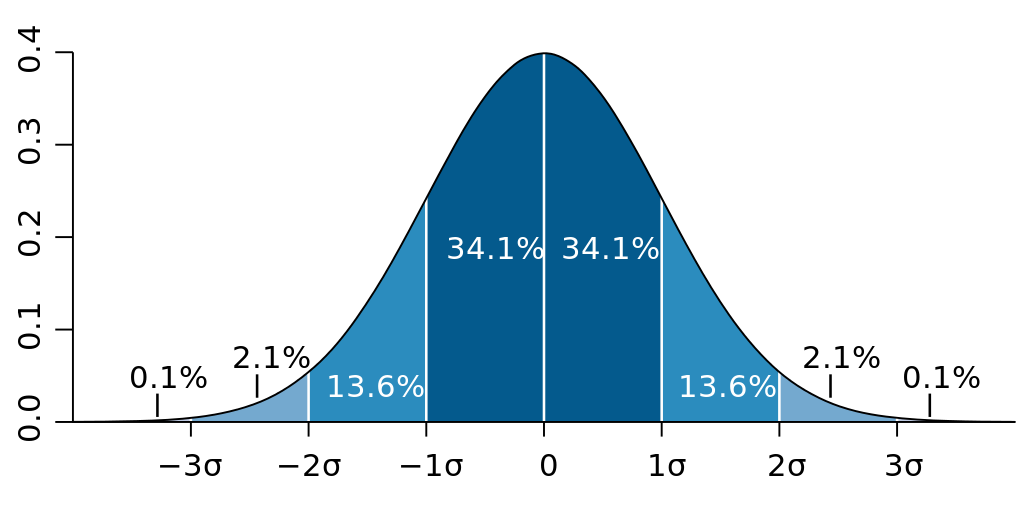
Usually when a B-tree predicts a position it will predict a node which will have numerous keys in it (kind of like 2-3-4 trees). The number of keys in a given node is called a page size. Then it will search through that node to find the exact key you are looking for. For machine learning after the machine learning algorithm has trained on the data, you will give it a key and it will predict a position. Usually when working with machine learning it is not always 100% accurate so that is why there is a min and max error bound. If the predicted position is not the exact one you are look for one approach will be to search through the area between the predicted position – error and predicted position + error. You find the error by calculating what was the worst prediction that was made subtracted by the actual value. This will give it a range to look through now. Something similar to the B-tree of finding a node and searching through that node to find the exact position.

To get the best results you need the data to be overfitted. This is the opposite mindset of usual machine learning experiments. You need the data to be overfitted because it will then give you the best predictions on the data. You also really only care about the data you do have. Overfitting in this case is good. B-trees essentially overfit the data too, which is what you want.

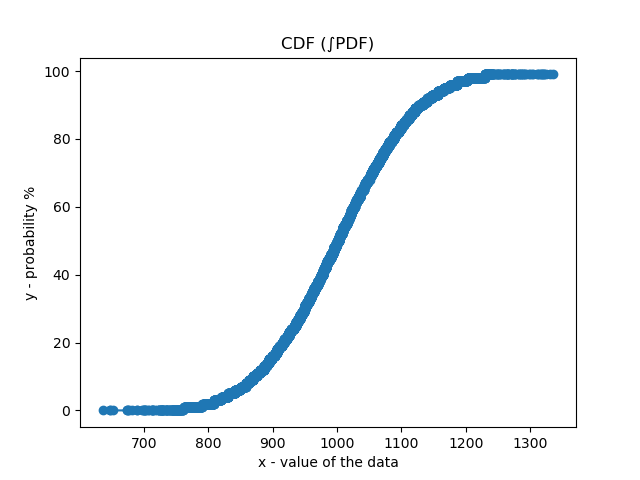
In a more mathematically sense for the case of a B-tree (considered as a model) it is predicting the cumulative distribution function (CDF). The better it can “predict” the CDF the more accurate your prediction. The CDF is the probability of a value occurring at the x-value you are predicting and then all of the x-values before. For example, if we have your probability density function (PDF) as so. The integral of a PDF is the CDF. A PDF is the probability of a given point happening of the data set.

Figure 2.

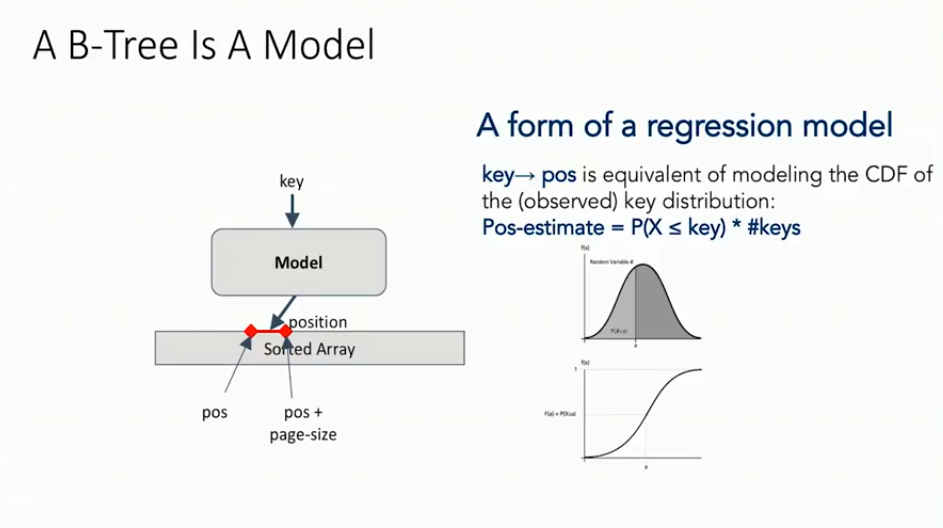
This is the PDF of a normally distributed dataset. It is also known as the “bell curve”. This means that there is a low probability of 700 happening, and a high probability of 1000. In your statistics class you might have seen something like this, which is a PDF of a normal function.

Figure 3.

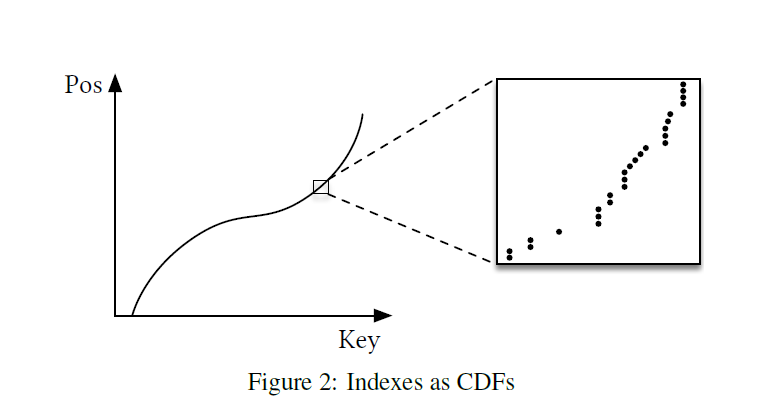
However, you cannot really make predictions of a data point using PDF because there is infinite amount of points between 900 and 901. You need to take the integral of the PDF to get the CDF.

Figure 4.

From there you can predict the probability of the value of 1,000 occurring, which is about less than or equal to 40%. If you multiply that by the number of keys or data points you have, it will be an estimate about that key’s position. As an example, let’s find the position of key 1,000. It is less than or equal to 40% and there are 3,000 keys. 0.4 \* 3000 = 1,200. The estimated position of key 1,000 is 1,200.

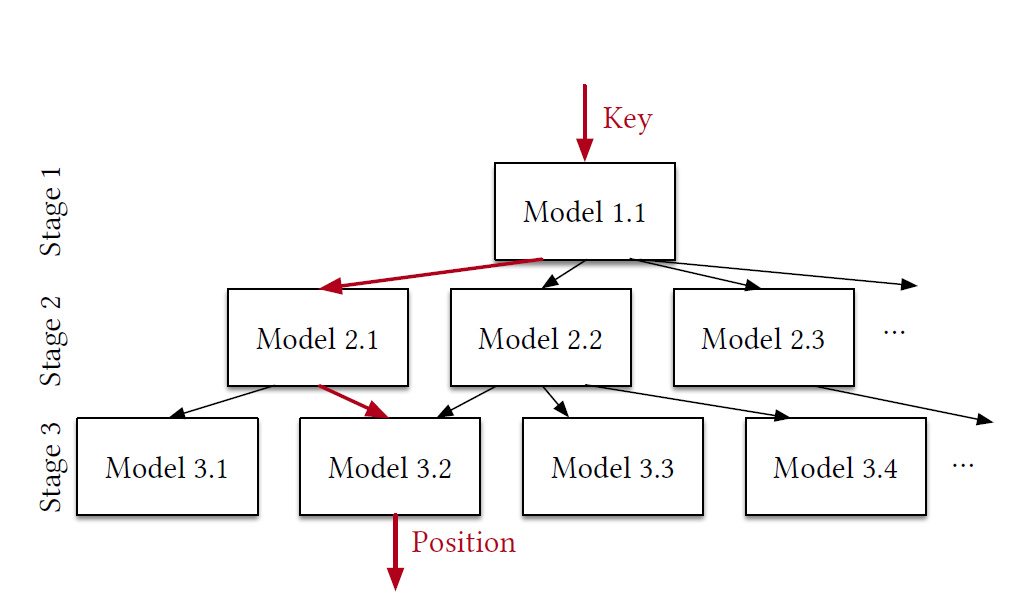
Figure 5.

This works fairly well however, learning the entire distribution all at once might not produce the best results. It is really good at giving an estimate but, not an exact answer. This is because it is really difficult to find the function that will predict every data point exactly.

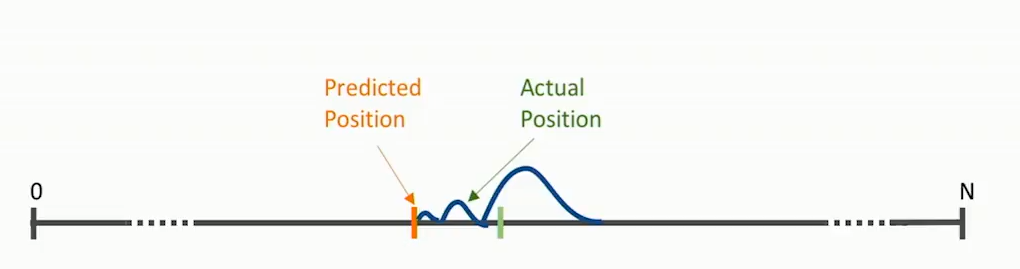
Figure 6.

For example, it is probably fairly easy for a machine learning algorithm to produce the function on the left-hand side of the figure. However, there might not be a function to predict every point correct such as when we zoomed in on the data to the right. For predicting the position, you need to get the correct position otherwise you will access a different key from what you initially wanted. In the research paper it is called the last mile problem.

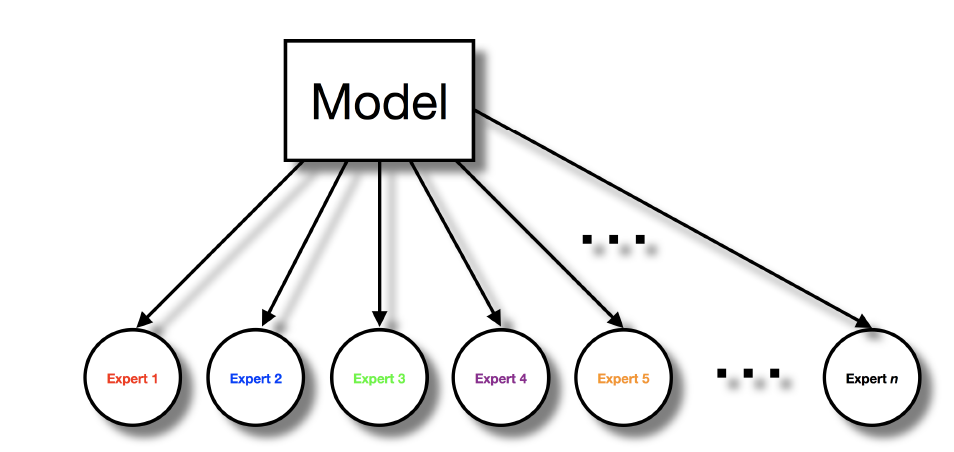
To solve the last mile problem a recursive model index (RMI) was built. The basic idea behind it is that you have models predicting models.

Figure 7.

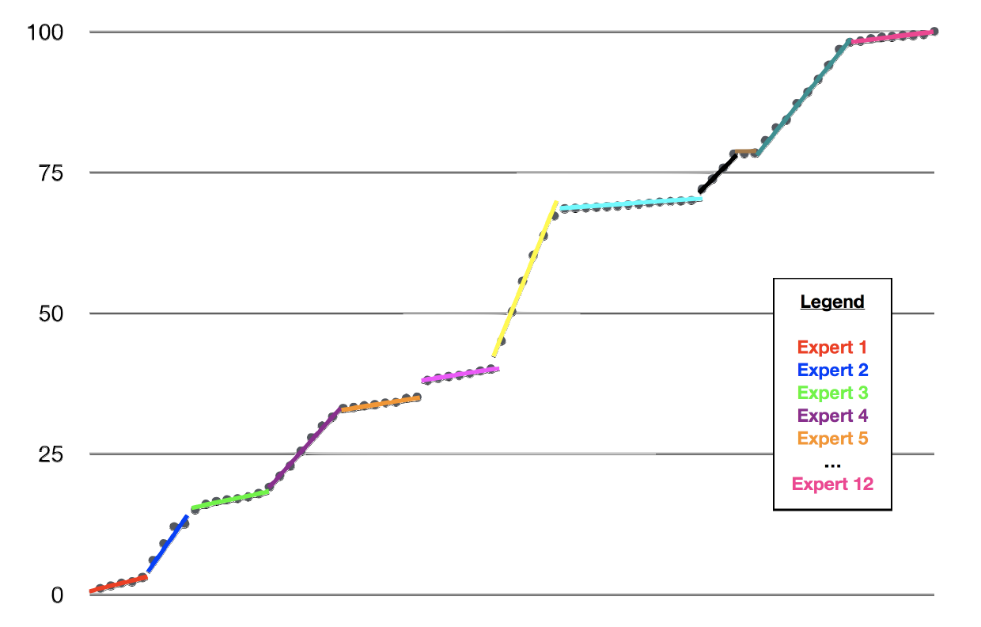
The idea is that there will be one model at the top (Stage 1) which will take in a key. That model will contain the entire data set. It will predict a position based on the entire data set. Based on that model’s predicted position it will send it to another model (Stage 2) which will have a subset of all the data, containing that predicted position (and hopefully the actual position of the key). Due to that model having less data to work with it will be more accurate to find the actual position. You can decide how many models you want in each stage (besides Stage 1 which will only have 1 model). That model will then send it to another model (Stage 3, following that same process as above), which will give you the actual position of the key back. Stage 3 also includes that model searching for the predicted position and then searching by a technique of choice to find the actual position. The searching technique used in the paper is exponential search. This is because you want to take advantage of where the model predicts the position at or to keep locality. This is because maybe the predicted position was only 1 off. If you used a binary search with the data contained in that model then you would lose locality. What exponential search is, if the predicted position in lower then you increase the predicted position up 1, then next time 2, then 4, then 16, etc. Until you found the actual position. It is basically (previous position increase) ^2. If it overshoots then it will do the same process will negative number. -1, -2, -4, etc. Graphically explained below.

Figure 8.

To graphically explain the RMI.

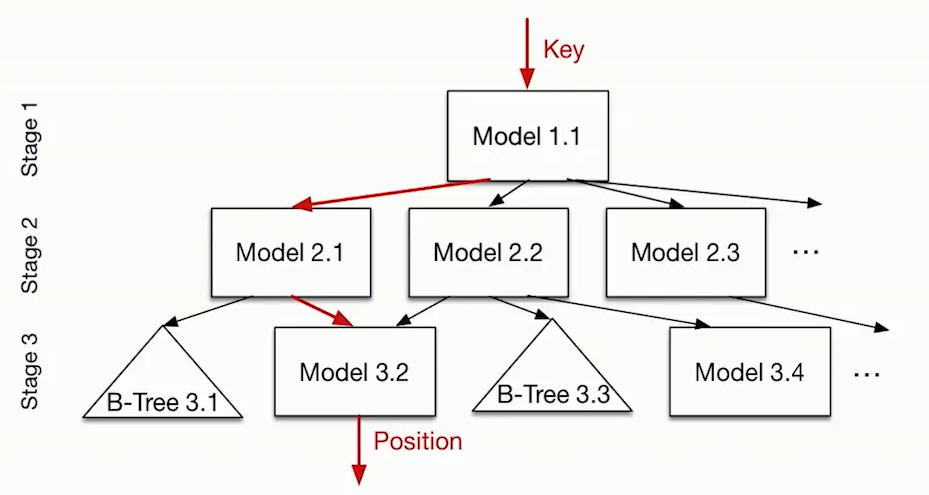
Figure 9.

You have your main model. In the research paper they used a neural net with 2 layers consisting of 32 nodes using the relu activation function. Which will then send the neural network’s predicted position to another model with a subset of data. These models will be experts on a portion of the data set.

Figure 10.

Then stage 2 can be another machine learning model. It can be whatever will best fit your data. Usually if you pick enough experts using a linear regression will work well. The number of experts or models in stage 2 is up to you. The data will also be split proportionally into how many experts or models you have.

To make an improvement on the RMI. The Hybrid RMI was made.

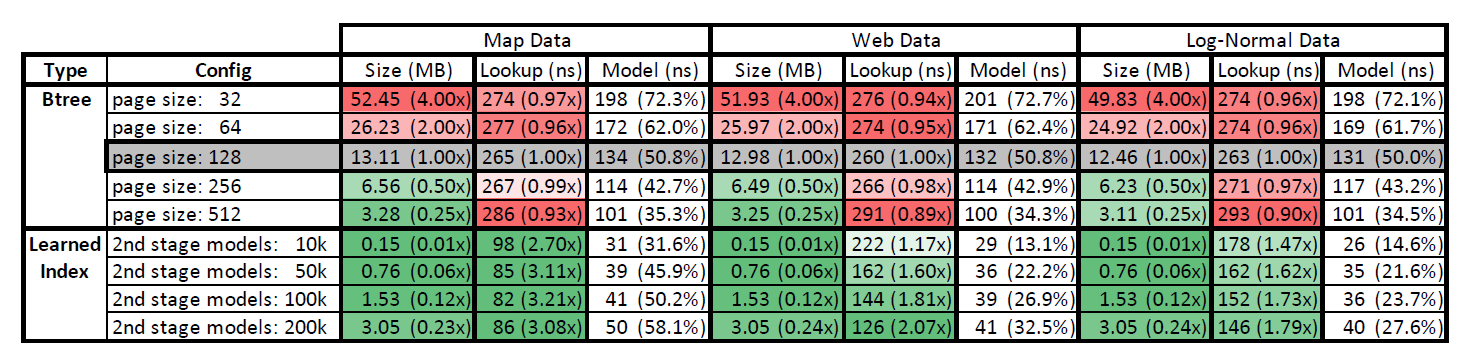
Figure 11.

Each model in the Hybrid RMI can either be a traditional machine learning algorithm or it can be a B-tree. This is decided by which model can predict the actual position faster, machine learning or a B-tree. This was done to ensure that the worst-case performance is that of a B-tree.

The potential advantages of using this approach rather than using the traditional B-tree are that it will use less memory, faster lookups, can take advantage of parallelism, take advantages of different hardware, and faster at inserting more data into the current data set.

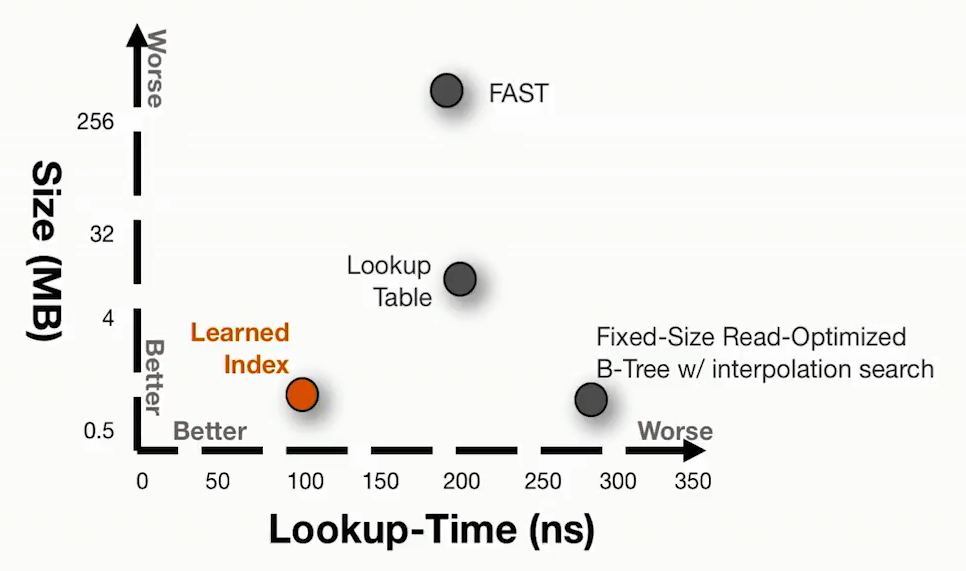
The meaning of smaller indexes is that you will have less data to remember to correctly index the data. For example, besides the data for a B-tree you need to include all the workings behind creating the B-tree (space complexity of a B-Tree is O(n)). If a linear model can accurately predict the position all you would need is the values for the intercept and the slope (y=mx+b). The linear model will also be faster at looking up the position because it is just calculating y=mx+b, and not traversing through the tree which is a search complexity of O(log n). A potential reason that it will be faster is that it can take advantages of parallelism because you are just solving multiplication problems or other mathematical problems that can take advantage of using parallelism. Which can take advantages of GPUs or TPUs. While if statements to traverse through a B-tree has to be done sequentially and through CPUs. There is a possibility of cheaper inserts. You might not need to train the model every time you input another data point in the dataset. You can still use the same models as before. While a B-tree you have to restructure the whole tree which is O(log n). This is another talking point because at some time after so many inserts you would want to retrain your models with the updated dataset so that it is more accurate. This will take time. What the best way to go around this was not part of the scope of the research paper. A solution they did give was to retrain the model when you think you need overnight.

The results after optimizing the machine learning algorithms were great. They used two staged RMI. With a neural net for the top model. They used a grid-search to find the best parameters. They tested from 0 to 2 hidden layers consisting of 8 or 16 nodes. With the non-hidden layers from 4 to 32 nodes. The second stage of the RMI consisted of linear models (which they found to give the best results). They did not use hybrid RMI. They also used binary search at the end because they did not find significant improvement over the other searching algorithms.

Figure 12.

To give you the most accurate description about each data set, I took a snippet of the actual research article. “The Weblogs dataset contains 200M log entries for every request to a major university web-site over several years. We use the unique request timestamps as the index keys. This dataset is almost a worst-case scenario for the learned index as it contains very complex time patterns caused by class schedules, weekends, holidays, lunch-breaks, department events, semester breaks, etc., which are notoriously hard to learn. For the maps dataset we indexed the longitude of ≈ 200M user-maintained features (e.g., roads, museums, coffee shops) across the world. Unsurprisingly, the longitude of locations is relatively linear and has fewer irregularities than the Weblogs dataset. Finally, to test how the index works on heavy-tail distributions, we generated a synthetic dataset of 190M unique values sampled from a log-normal distribution with µ = 0 and σ = 2. The values are scaled up to be integers up to 1B. This data is of course highly non-linear, making the CDF more difficult to learn using neural nets. For all B-Tree experiments we used 64-bit keys and 64-bit payload/value” [1].

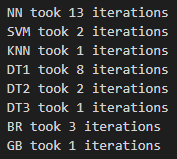
As a more general way of displaying the results.

Figure 13.

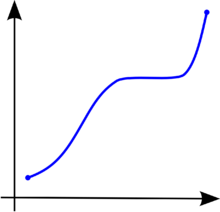
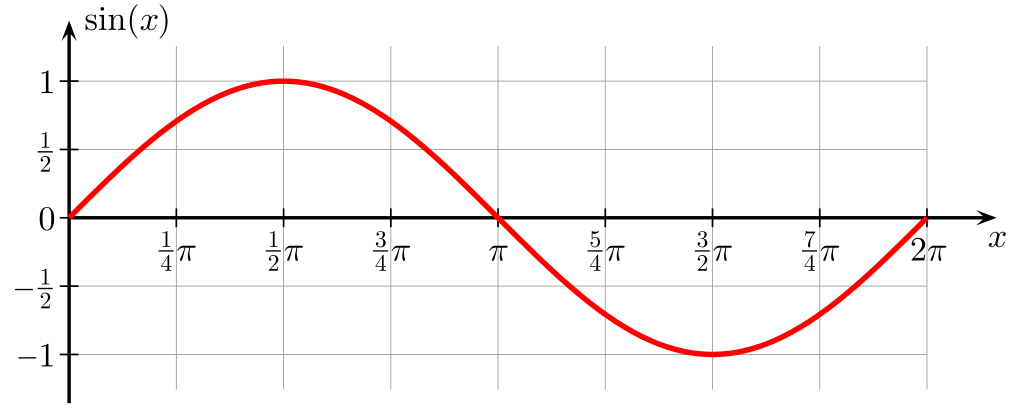
As you can see learned indexes does very well. However, this is not true for every data distribution. In order to see if the data distribution you want to index can achieve faster lookup times you would need to test it out. This is not meant to replace traditional index structures just provide a new way outlook on how to do things with machine learning. This has been done with bloom-filters, and hash maps. The researches think that machine learning has great potential in the work of learned algorithms such as sorting, and scheduling. Other future work can include learned joins, range-filters, priority queues, cache policy, and multidimensional arrays.

**My findings on B-Trees**

I created my own simple implementation of this concept. I got the idea to do it that way by yash21saraf on Github (provided a good starting point to add on to). Basically, the code that I produced creates the data, graphs it (CDF, PDF, and a plotting it as a regular line graph), and then puts the data into a csv file. It uses, x (key) and y (position) variables instead of just the keys (a one-dimensional array), then calling or creating a function to get the position (if you do it this way the keys have to be sorted). It was done that way because it was easier for me to program and to see what was happening. The function that was tested was x = y^2 + 21\*y + 34. Then for the top model, I wanted to see which machine learning algorithm did the best. I tested a neural network for regression, SVM (did a grid search to find best parameters), KNN (1 neighbor, K neighbors regressor), three different decision trees (max depth of (DT1),5 (DT2), and 10 (DT3)), Bayesian ridge, and gradient posting (more information on the exact parameters are used can be found on my Github). Then I split the data into 100 different sequential parts (or buckets). For stage 2, it is 100 linear regression models with each linear regression model with 1/100 of the data. After the linear regression model predicted a position, I used exponential search to find the actual position.

Figure 14.

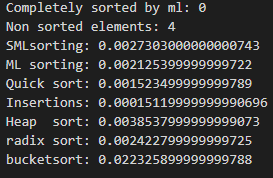
To my surprise the neural network did the worst. The number of iterations is how many iterations of exponential search went through. If it was 1 iteration that means the linear regression model predicted the actual position correctly. It did not work really well with a function that was not monatomic (entirely increasing or decreasing) such as sine.

Figure 15. Vs. Figure 16.

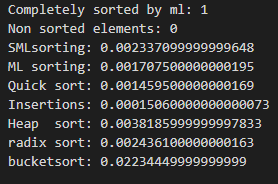
This is because in my code I took the ceiling of all the predications to make it an integer because all of the keys are integers. It would also make exponential search more complex the numbers were floats and not integers. Another reason is that you might have two keys that are the same value in different indexes which will confuse the models and might give you the wrong position.

**Learned Sorting**

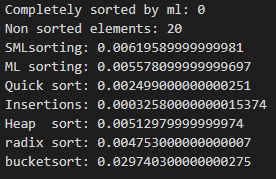
The idea behind learned sorting is that based on the x value a machine learning algorithm will try and predict the y value for that x value. The function that I tested it on was x = the ceiling of 2^3 + 4^2 + 3(y) \* log(y). The list has 1,000 items. I shuffled the data with the built in python function random.shuffle() then I put the data into an csv file. Reshaped the data from -1 to 1. Then I used K neighbors regressor (used because I thought it is one of the faster machine learning algorithms to train and predict) with number of neighbors equal to 1 (produced the best results). How it works is that the machine learning algorithm gets each x value and predicts a y position to put into a new list. If there is another x value with the same y position predicted value, a placing algorithm will place these x values into the sorted list after the rest of the x values have been placed. After that, all those x values that could not get placed will get placed by checking the next position up (based on the predicted position) to see if it filled. If it is not that x value will get placed there. If not, it will check one lower than the predicted position, then predicted position +2, predicted position -2, etc. It will keep doing this until it finds an empty slot to place the x value in. After all every x value gets placed into the list. Insertion sort is used to sort the list after that. Insertion sort is used because it does well with nearly sorted lists. If the machine learning algorithm gets the list sorted perfectly then it will not need to go through placing the x values that had its spot taken. Then I tested it against a handful of other sorting algorithms.

Figure 17.

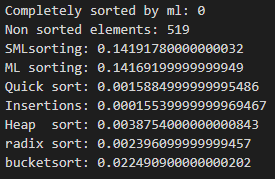
If the first line is 0 that means the machine learning algorithm did not sort it flawlessly. It is 1 then it did. SMLsorting starts the timer when the machine learning algorithm starts it training. ML sorting means that the timer starts when it starts predicting. The sorting algorithms in order are quick, insertion, heap, radix, and bucket sort. All of these times are in seconds. As you see the results look very promising. ML sorting is usually as about middle or a little bit higher up in the pact.

Figure 18.

The times when K neighbors regressor sorted it without any extra help (on the same dataset). This time was neighbors =1 and the previous results were neighbors = 5.

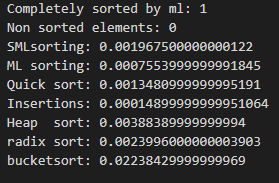
Figure 19.

The results when there are 20 mis predicted items.

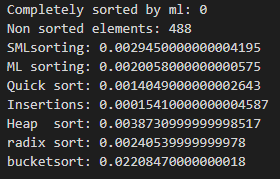
Figure 20.

Neighbors = 500. The time when the machine learning algorithm predicts really poorly.

Testing it out with a decision tree with max depth of 10 was extremely promising.

Figure 21.

ML sorting is now the second lowest.

Figure 22.

This is when the decision tree has max depth of 9. Even when it mis predicts about half of the list it is still 3rd lowest in time, which is really amazing. Learned sorting using decision trees cuts down on time compared to k nearest regressor.

**Conclusion**

The area of using machine learning to enhance traditional data structures, and algorithms looks like a really promising field that hopefully will yield a lot of fruit.

**Figures**

Figures 1, 6, 7, and 12 are from the research paper, “The Case for Learned Index Structures”: <https://arxiv.org/abs/1712.01208>

Figures 5, 8, 11, and 13 are from the presentation from the above paper: <https://www.youtube.com/watch?v=qFbw4MgR6pI&t=1582s>

Figures 9, and 10 are from the research paper, “Learned Hash-Index: A Collision Competition”: <http://stanford.edu/~jdoan21/cs166paper.pdf>

Figures 3, 15, and 16 are from Wikipedia respectively:

<https://en.wikipedia.org/wiki/Standard_deviation>

<https://en.wikipedia.org/wiki/Monotonic_function>

<https://en.wikipedia.org/wiki/Sine>

Figures 2, and 4 are graphs that I made using the normal dataset from daravinds Github repo:

<https://github.com/daravinds/Learned-Index-Structures>

Figures 14, and 17-22 are graphs that I made

**Great Resources to learn about Learned Indexes:**

GitHub repositories:

<https://github.com/mikebenfield/Learned-Index-Structures>

<https://github.com/daravinds/Learned-Index-Structures>

<https://github.com/amorten/openlis>

<https://github.com/thaonguyen19/Learned_HashMaps> (Github repo for the third research paper)

<https://github.com/Learned-Index-Structure/LIS-Training> (yash21saraf’s repo: the one I based by learned B-tree on)

<https://github.com/learnedsystems/SOSD> (Github repo about the second research paper)

<https://github.com/TheWifflebrain/LearnedIndexStructures> (My Github repo, simple\_LIS.py is the learned B-tree and simple\_sorting.py is learned sorting)

Research papers:

[1] <https://arxiv.org/pdf/1712.01208.pdf> (The original research paper, the one this paper is based on)

<https://arxiv.org/pdf/1911.13014.pdf>

<http://stanford.edu/~jdoan21/cs166paper.pdf>

Lectures:

<https://www.youtube.com/watch?v=qFbw4MgR6pI&t=1173s>

<https://www.youtube.com/watch?v=NaqJO7rrXy0&t=1130s>

<https://www.youtube.com/watch?v=0q9mxMekBeE&t=1252s>