大数据计算基础实验报告

1190201207 朴希锐

1. **绪论**

1.1研究问题的背景

索引是一种用于提升数据存取性能的辅助数据结构，目前已成为数据库系统不可或缺的技术 之一。传统的 B+树索引结构是基于磁盘存储设计的，即假设整个索引完全存储在磁盘中。 为了提升数据库系统的读写性能，现代数据库系统倾向于将整个索引缓冲在内存中。当数据库规模较小时，缓存索引是可行的，也可以获得良好的性能。然而，在大数据时代，数据量 达到了 PB 级以上，由于传统的 B+树索引结构具有 O(n)的空间复杂度，将导致索引无法全部缓存到内存，进而影响查询性能。近年来，人工智能、机器学习技术的快速发展引起了数据库领域的重视。如何借助人工智能、机器学习技术来优化数据库索引等传统数据库技术，成为当前数据库领域关注的热点问题。MIT 的研究人员 Kraska 等人在 2018 年 SIGMOD 会议上 首次提出了学习索引(learned index)的概念。 为了解决初始版本提出的学习索引的一些缺点，Tang 等人提出了并发的学习索引。

1.2论文实现用到的背景知识

1.2.1什么是索引：

* 索引可以帮助我们在海量数据中快速定位到某些特定条目。
* 尤其是当数据并没有按照我们要查询的字段排序时会非常有用。

假设我们存储了大量数据，当我们需要查询其中某些特定数据时，我们并不希望从头到尾扫描整个磁盘记录来寻找该数据，例如，当我们查询 “名字叫 Tim 的人有哪些” 或者 “某个特定日期范围内出生的人” 等，我们可以使用索引。例如哈希表、B 树等等，我们将在后面对这些传统索引以及新型的学习索引进行对比讨论。

1.2.2索引性能的评价：

对于索引结构，我们一般会从以下几个方面考察其性能优劣：

* 查询类型 (单值查询 VS. 范围查询)
* 查找速度
* 插入一条新记录的开销
* 删除一条记录的开销
* 更新一条记录的开销
* 索引大小和利用率

但是请注意，在Kraska 等人的论文中并没有显式地考虑插入、删除、更新记录的开销，论文《The case for learned index structures》主要集中在只读数据集，因此并没有涉及到这些操作。所以实际上，这是其提出的多层神经网学习索引存在局限性的一个方面，将在第二篇论文中进行讨论和解决。

1.2.3 B树：

对于范围查询，最常见的索引结构是 B 树，其结构和二叉树非常相似，区别在于 B 树具有分支因子。

B 树作为索引，在性能方面具有以下特点：

* 非常适用于范围查询，同时也很适用于单值查询。
* 查询时间复杂度为 O(logn)，叶子结点包含了我们实际需要的数据，并且树的深度为 logn。
* 插入和删除的时间复杂度为 O(logn)，和查询一样。
* 索引大小和利用率取决于每个结点的大小、树的整体平衡度以及每个结点的饱和程度。

1.2.4 学习索引:

学习索引的含义是，范围索引可以被看作是将键映射到数据位置的函数。对于固定长度的键值记录，假设它们在数组中排序，这个函数实际上是键分布的累积分布函数(CDF)。给定CDF F，记录的位置是⌊F(键)× N⌋，其中N是记录的总数。学习索引背后的核心思想是使用机器学习模型(如深度神经网络)近似CDF，并使用模型预测记录位置。为了保证模型的正确性，学习索引存储了模型的最大和最小预测误差。通过计算每个键的实际位置和预测位置之间的差来计算误差，并存储最大值和最小值，对于排序后的数组中的记录，它的实际位置必须是[pred(key) + min\_err, pred(key) + max\_err]，其中pred(key)是预测的位置。

1.2.5学习索引的限制:

学习索引具有性能优势，但有两个问题限制了它的实用性。首先，学习到的索引没有提供有效的方法来处理写操作，特别是在并发场景下。基于当前的设计，一个直观的解决方案是在delta索引中缓冲所有的写操作，然后用学习到的索引周期性地压缩它。压缩包括将数据合并到一个新的已排序的数据数组中，并重新训练模型。尽管简单明了，但这种方法在查询时存在严重的速度放缓问题。其次，学习到的索引的性能与工作负载特征密切相关，包括数据和查询分布。这是因为查找效率取决于为查询激活的特定叶阶段模型的错误边界。同时，不同模型的误差范围也不同。因此，在特定的工作负载下，学习到的索引可能比b -树的性能更差。

1.3本文的工作要解决的问题

本实验的实现分为两个部分：

1. 第一部分是实现MIT 的研究人员 Kraska 等人在 2018 年 SIGMOD 会议上提出的论文《The case for learned index structures》中的学习索引方法，同时自己设计其他不同于此论文中利用多层神经网实现的学习索引方法，如B-Tree，Linear Regression，K-Nearest Neighbor，Naïve Bayes，Decision Tree等机器学习索引方法并与论文中神经网Neural Networks学习索引的实现进行对比；
2. 第二部分是实现Tang 等人在论文《XIndex: A scalable learned index for multicore data storage》中提出了并发的学习索引XIndex，这种新的学习索引可以支持数据更新，不同于现有学习索引的更新方法。

1.4当前研究工作的不足之处及挑战

1.4.1 论文《The case for learned index structures》中的不足之处及挑战：

到目前为止，此论文中所有生成的结果集中在只读内存数据库系统的索引结构上。即使没有任何重大修改，当前的设计已经可以替代数据仓库中使用的索引结构，这些索引结构可能每天只更新一次。

(1)插入和更新

初看起来，由于学习模型潜在的高成本，插入似乎是Kraska 等人设计的初代学习索引的致命弱点，但是学习索引对于某些工作负载可能具有显着的优势。 一般来说，我们可以区分两种类型的插入：（1）appends 和（2） inserts in the middle，例如在订单表上更新客户id上的二级索引， 现在我们关注后者，并考虑在我们的已排序数据集中引入额外空间的方法，类似于B树通过其每个页面的最小和最大填充因子在已有页面中引入额外空间。 最后，假设插页大致遵循与学习的CDF相似的模式。在这些假设下，模型可能根本不需要再培训。 这样，对新项目的插入，学习索引可以“概括”成为O(1) 操作，因为它们可以直接放入可用空间中，并且空间可用于最常用位置。 相反，B-树需要O (log n)用于查找和重新平衡树的插入操作。

处理插入的另一种更简单的方法是构建一个增量索引和许多其他系统中，并且还具有如下优点：对于大型再训练操作，可以使用诸如GPU / TPU的专用硬件，这将显着加速，即使需要对整个模型重新训练也是如此，这也就是第二篇论文中解决使用的方法。

（2）分页

在此文中，我们假定数据（实际记录或<key，pointer>对）存储在一个连续的块中。 但是，特别是对于存储在磁盘上的数据的索引，将数据分区为存储在磁盘上的不同区域中的较大页面是相当常见的。 为此，我们观察到一个模型学习CDF不再成立，下面我们概述几个可以解决这个问题的方法：  
利用RMI结构：RMI结构已经将空间划分为区域，通过对学习过程的小修改，我们可以最大限度地减少它们覆盖的区域中模型的重叠程度。 此外，可能会复制任何可能被多个模型访问的记录。 这样我们可以简单地将偏移量存储到模型中，也就是数据存储在磁盘上的位置。另一种选择是以<first\_key，disk-position>的形式提供额外的转换表。

1.4.2 论文《XIndex: A scalable learned index for multicore data storage》中的不足之处及挑战：

(1)内联值与分隔值

XIndex将键和值连续存储在data\_array中(内联值)。另一种流行的方法是将值存储在单独的存储器中，并且只存储指针和键(分隔的值)。对于小值，我们的方法在减少DRAM访问方面有优势，因为键和值都可以驻留在一个缓存线中。对于大的值，分隔值可以减少压缩成本，因为只有指针会被复制。然而，这两种方法都需要两阶段压缩，因为记录的元数据(例如删除和锁定)应该内联，并且可能在压缩期间更改。因为分离元数据可能会导致较高的压缩成本，因为它需要对每个记录进行额外的内存访问，以检查值是否在逻辑上被删除。

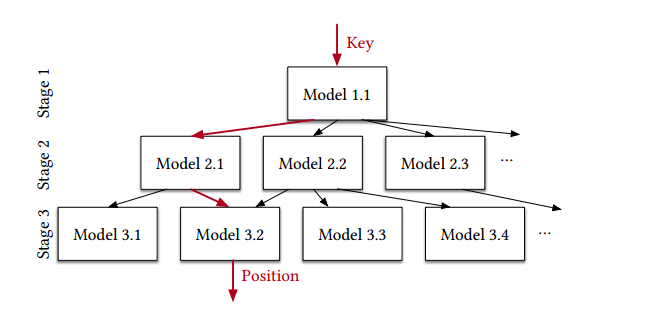
(2)局限性

首先，当数据集较小时，其他索引结构(例如stx::Btree)在模型计算成本和遍历两层结构的成本方面可以优于XIndex。其次，键长会大大增加模型训练和推理的开销，影响XIndex的效率。对于64字节的键，与8字节的键相比，性能下降了50%。我们将为可变规模的工作负载设计更灵活的结构，并将降低长键的成本作为未来的工作。

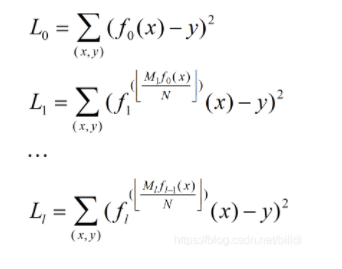
第二章 系统框架

2.1第一篇论文实现的框架：

第一篇论文实现了Kraska 等人在 2018 年 SIGMOD 会议上提出的论文《The case for learned index structures》中的学习索引方法-多层神经网络，用来代替所有现有的索引结构如B树索引，Hash索引等，作者指出：达到100M数据只有百级别的误检是比较困难的，但是降到万级别的误检是相较容易的，所以将误检从万级降到百级只需要将模型要处理的数据进行划分就行了，将B-tree的前两层替换为模型，提出了一种模型的垂直层次结构，即多层神经网框架如下：



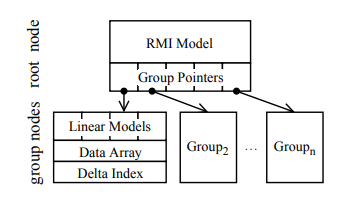
此神经网模型记为f(x),x代表key，y∈[0,N)代表预测位置，其中x是key，y∈[0; N）是位置，我们假设阶段i有Mi个模型。 我们在阶段0训练模型，f0（x）≈y。 因此，阶段i中的模型k，由f^k（k）表示，由损失函数训练：



迭代训练每个模型，每个模型都负责在一定错误率下得到预测值，然后根据预测值选择下一个模型，和上一个模型一样，这个模型负责降低错误率并传递于再下一个模型。

2.2第二篇论文实现的框架：

论文中实现的新的并发的支持数据更新的学习索引XIndex采用两层架构设计，顶层包含一个根节点，根节点对底层所有组节点进行索引。通过范围分区将数据划分为多个组。根节点使用学习过的RMI模型来为组建立索引。每个组节点使用学习过的线性模型来索引其数据。对于写操作，XIndex对现有记录执行原地更新，并将每个组与一个增量索引关联到缓冲区插入。XIndex引入了一种新的压缩方案——两阶段压缩，有条件地压缩增量索引。压缩在后台执行，不会阻塞任何并发操作。



第三章 第一篇论文及自己设计的其他学习索引算法及具体实现

3.1 B树模型训练算法：

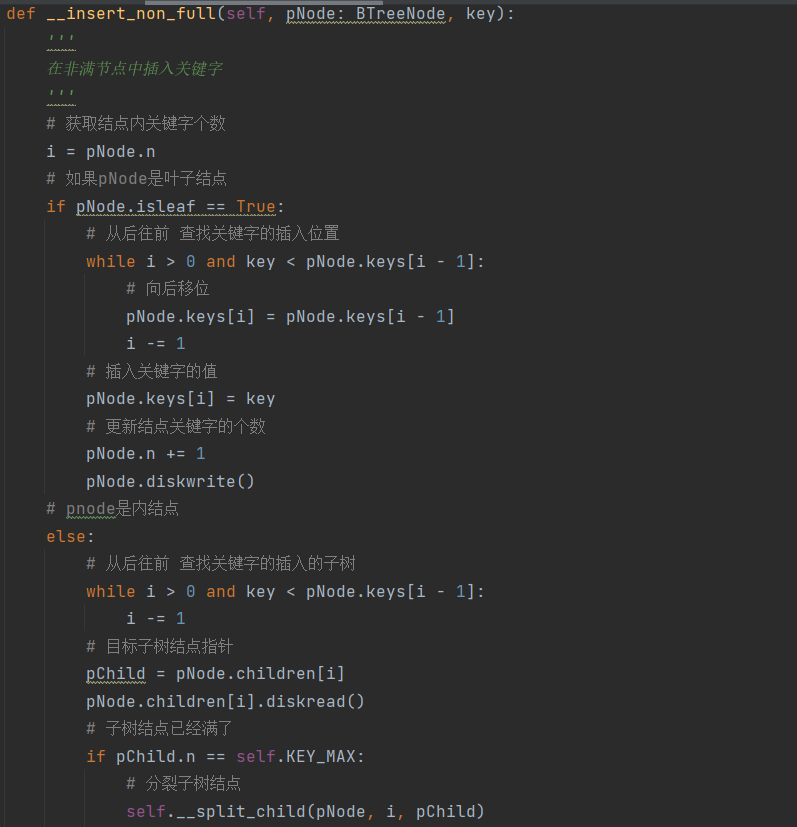
（1）首先将输入的数据按照key的顺序排好；

（2）对于输入的任一个数据记录，首先检查是否为空树，若为空树，那么创建B树模型的根节点root，将此记录的key和对应的位置放入根节点；

（3）然后检查根结点是否已满，若根结点数目已满，则创建新的一层的所有子树的根结点，根据收到的数据的key范围将其分配到对应的子树根节点上；

（4）更新节点指针直到分配完所有数据。

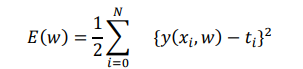




3.2线性回归模型训练算法：

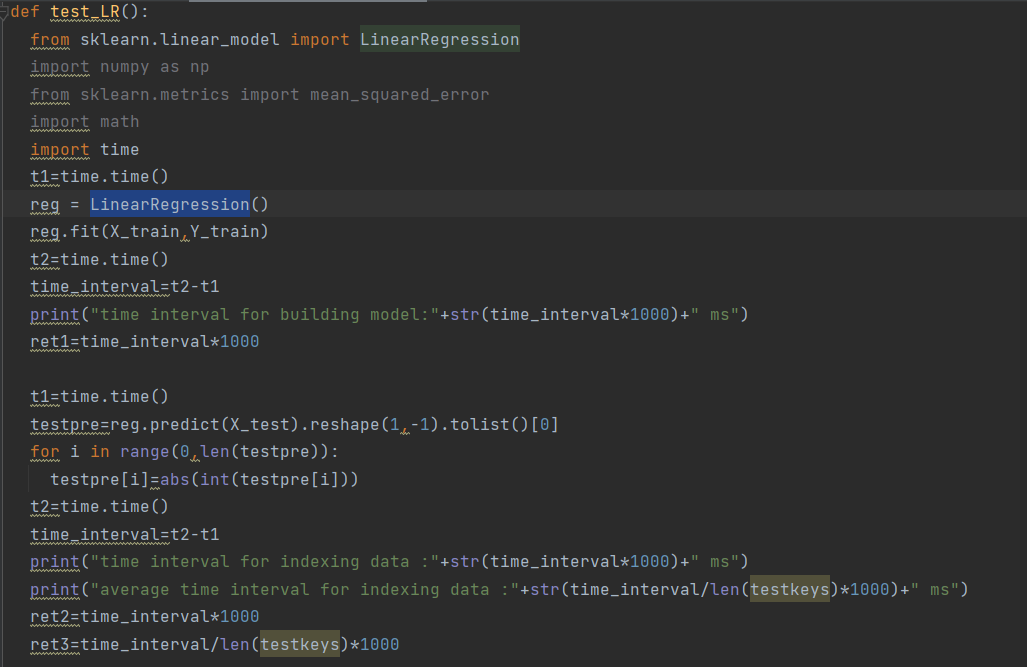
（1）因为我们的数据未必都是要过原点的，所以根据实际情况，得出模型 y = w \* x + b，其中x为每个数据的key，y为数据记录的位置；

（2）给定一个x，线性回归模型预测的值与真实的值的误差显示了模型的好坏，那么所有数据预测位置的偏差和为损失函数；



（3）通过对损失函数求最小值，也就是对损失函数求导为0得到最好的w；

1. 将w带入得到线性回归模型。



3.3 KNN（N个最近邻接点）模型训练算法：

在建立训练集时，就要确定训练数据及其对应的类别标签；然后把待分类的测试数据与训练集数据依次进行特征比较；从训练集中挑选出最相近的k个数据，这k个数据中投票最多的分类，即为新样本的类别。

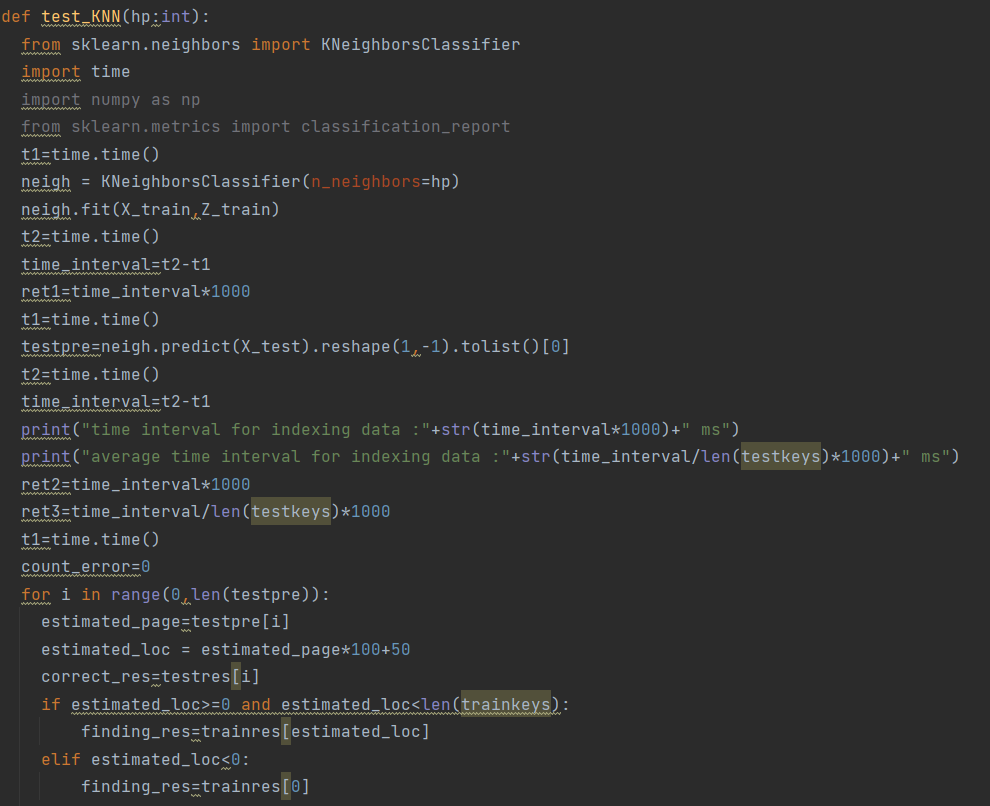
（1）计算测试数据与各个训练数据之间的距离；

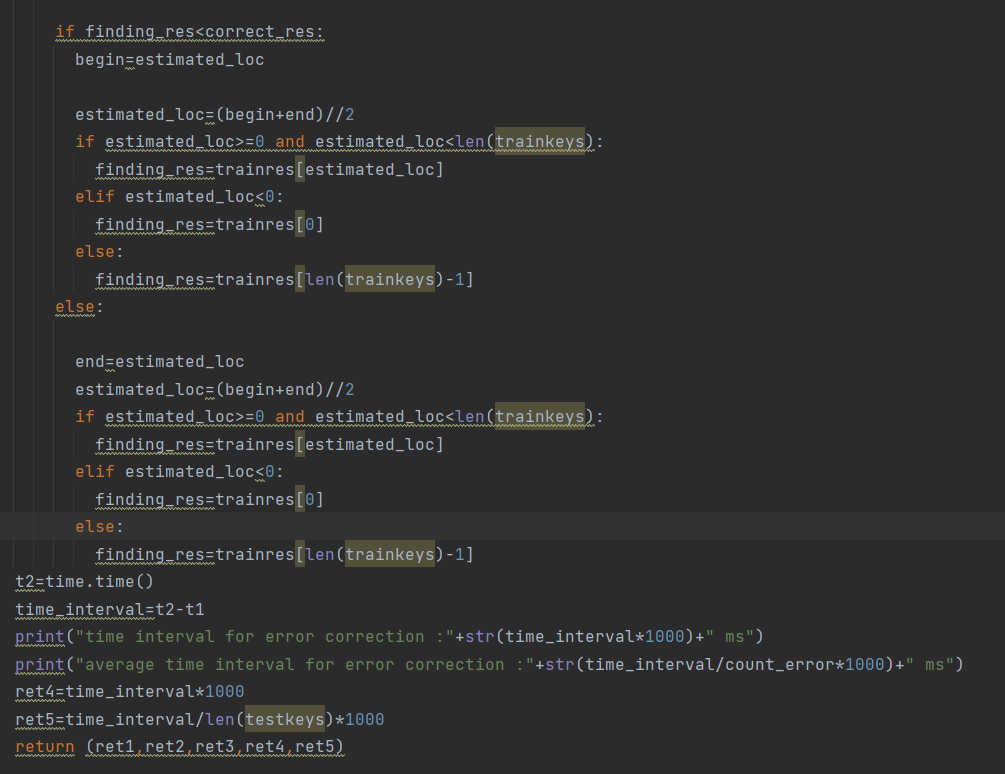
（2）对距离从小到大进行排序；

（3）选取距离最小的K个点；

（4）确定前k个点的类别出现频率；

（5）出现频率最高的类别作为预测分类；



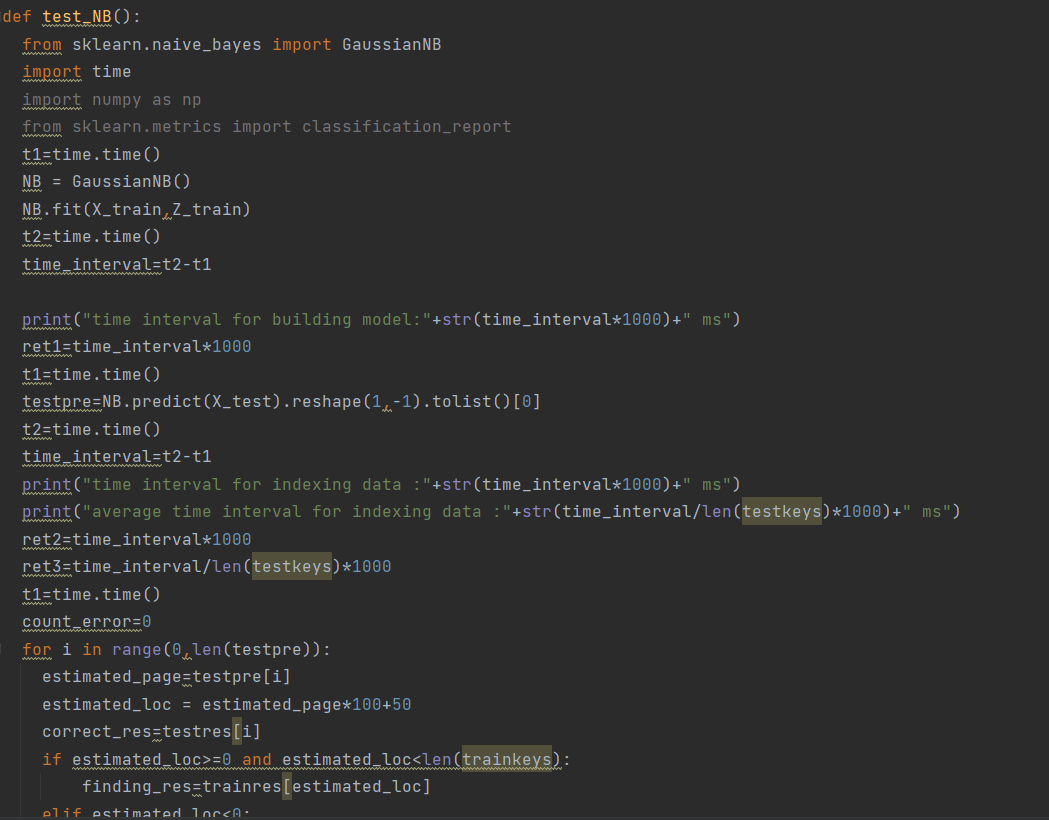


3.4 Naive Bayes（朴素贝叶斯）模型训练算法：

（1）根据具体情况对输入数据的key属性进行适当划分，然后由人工对一部分待分类项进行分类，形成训练样本集合；

（2）计算每个类别在训练样本中的出现频率及key属性划分对每个类别的条件概率估计，并将结果记录；

（3）最终得到一个朴素贝叶斯模型， 使用分类器对待分类项进行分类，其输入是分类器和待分类项，输出是待分类项与类别的映射关系。

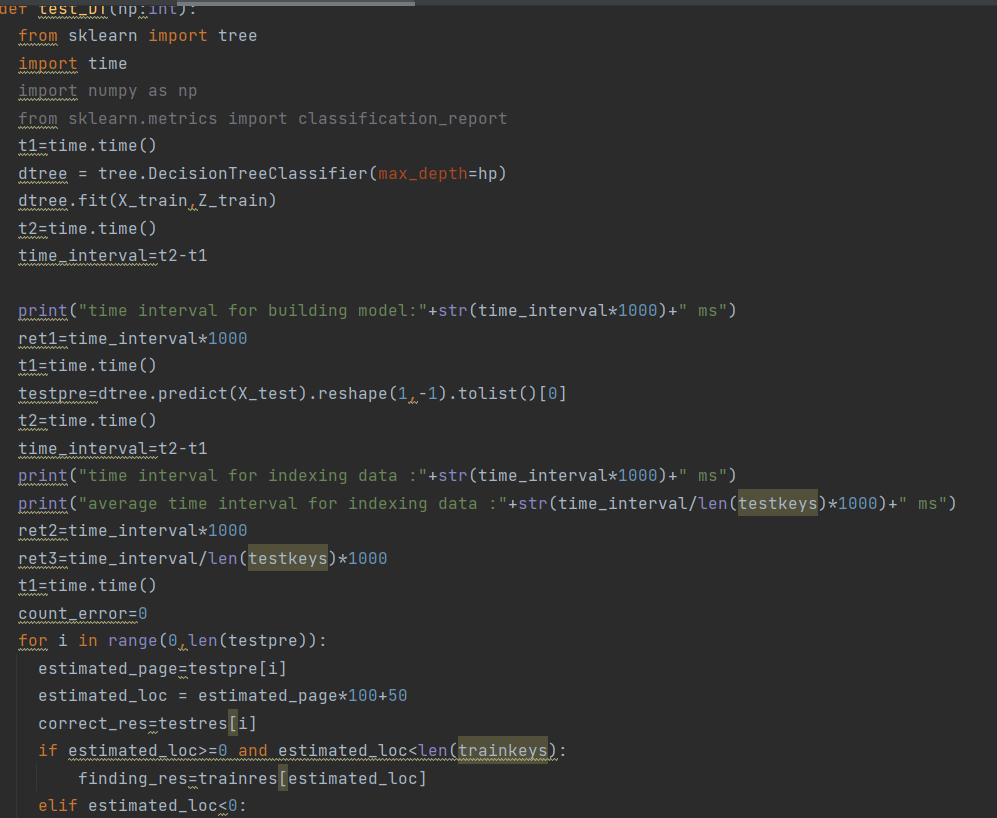


3.5 Decision Tree（决策树）模型训练算法：

（1）将所有数据放在根节点；

（2）选择一个最优特征，根据这个特征将训练数据分割成子集，使每个子集拥有当前最好分类；

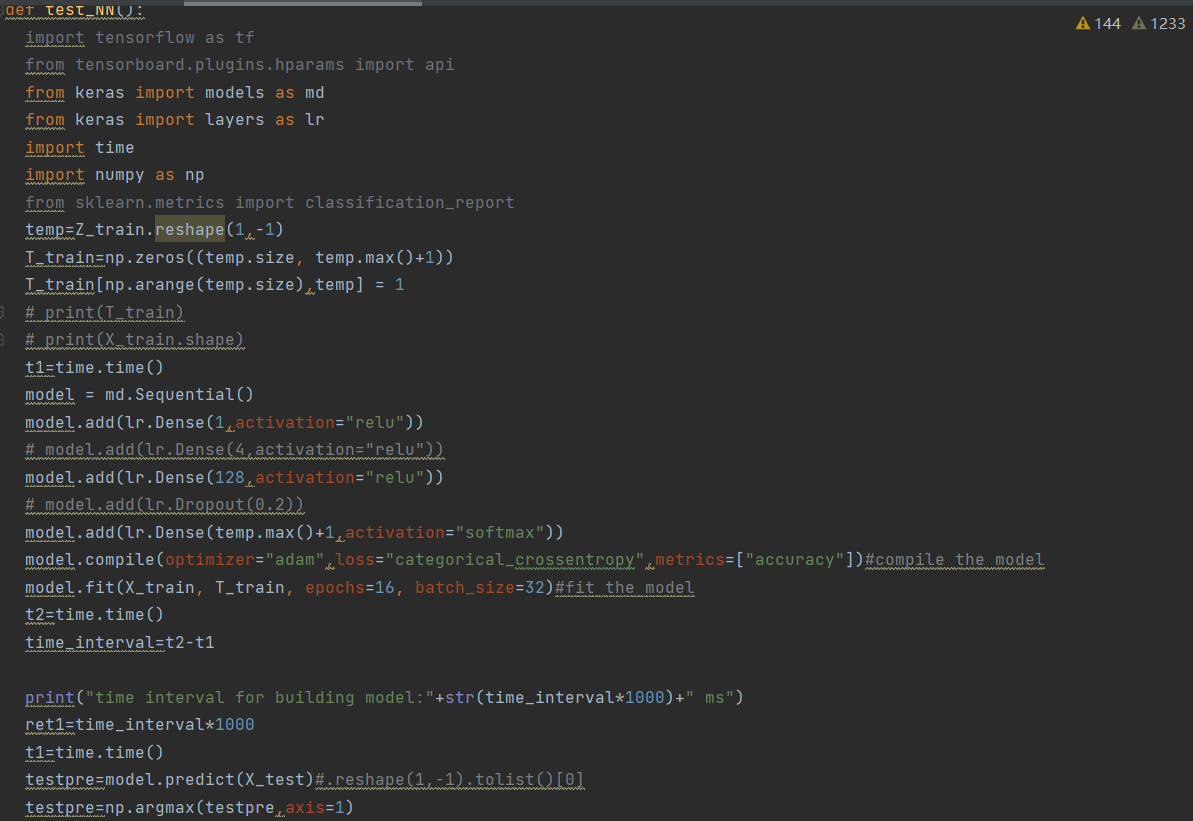
（3）递归下去，直到所有数据被基本正确分类。



3.6 Neural Networks（神经网）模型训练算法：

本论文中的神经网络，具有0到2个完全连接的隐藏层和ReLU激活函数，以及多达32个神经元和B树（也就是决策树）的层宽度。 给定一个索引配置，它指定阶段的数量和每个阶段的模型数量作为一个数组，混合索引的端到端训练按照以下算法完成：

（1）M = stages.size;  
（2）tmp records[][];  
（3） tmp records[1][1] = all data;  
（4）for i <- 1 to M do  
（5） for j <- 1 to stages[i] do  
（6） index[i][j] = new NN trained on tmp records[i][j];  
（7）if i < M then  
（8） for r ∈ tmp records[i][j] do  
（9） p = f(r.key) / stages[i + 1];  
（10） tmp records[i + 1][p].add(r);  
（11） for j <- 1 to index[M].size do  
（12）index[M][j].calc\_err(tmp records[M][j]);  
（13）if index[M][j].max abs err > threshold then  
（14） index[M][j] = new B-Tree trained on tmp records[M][j];  
（15） return index;



第四章 第二篇论文并行学习索引XIndex部分算法的实现

4.1 Structures：

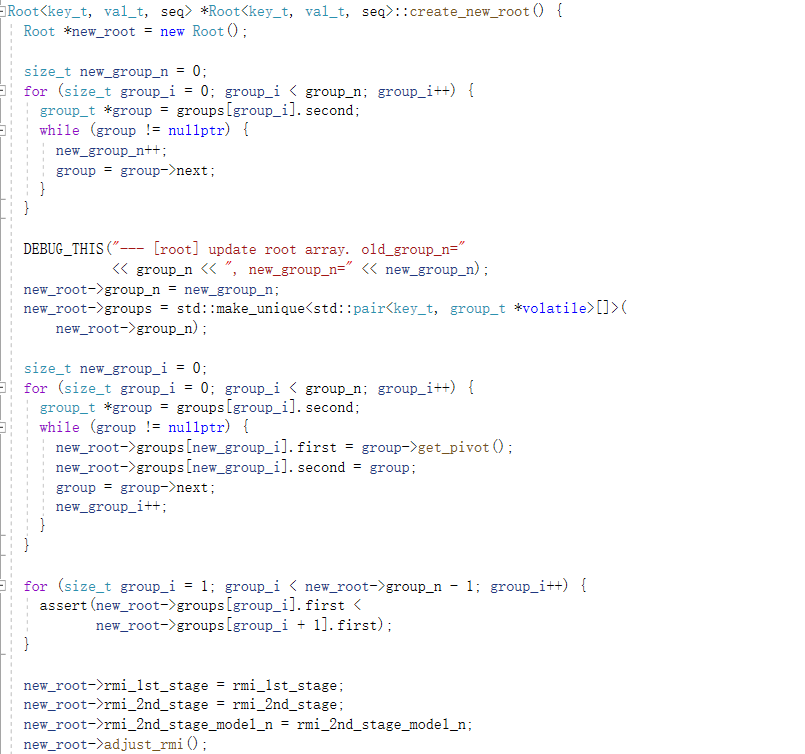
（1）struct root\_t:

rmi\_t rmi;

uint32\_t group\_n;

key\_t pivots[];

group\_t\* groups[];



（2）struct record\_t:

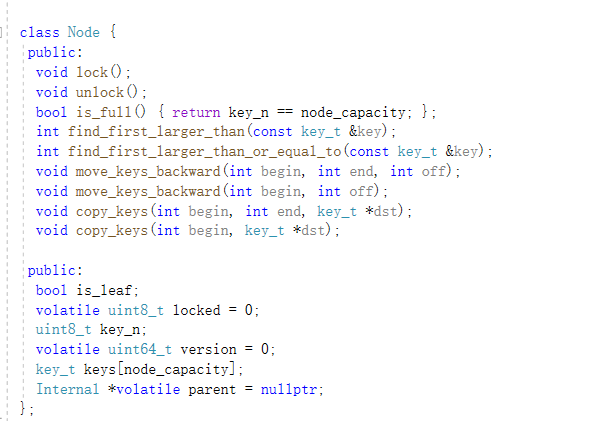
key\_t key;

val\_t val;

uint64\_t // composite 8B

is\_ptr : 1, removed : 1

lock : 1, version : 61;



（3）struct group\_t:

key\_t pivot;

bool\_t buf\_frozen;

uint16\_t model\_n;

uint32\_t array\_size;

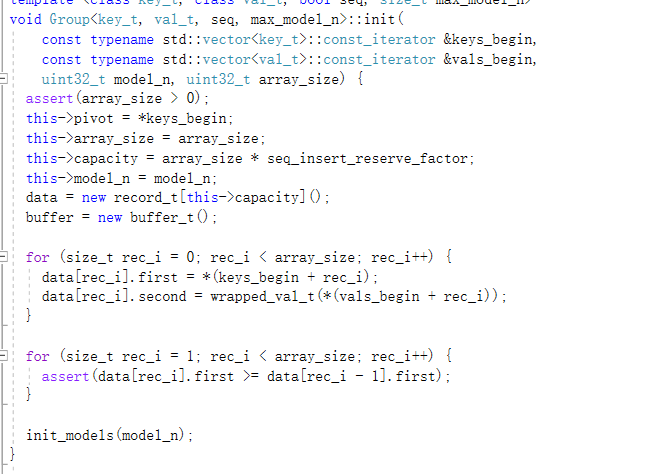
model\_t models[MAX\_MODEL\_N];

record\_t data\_array[];

buffer\_t\* buf;

buffer\_t\* tmp\_buf;

group\_t\* next;



4.2 Get and Put：

（1）get(root, key):

group ← get\_group(root, key)

pos ← get\_position(group, key)

val ← empty

if pos !=empty

val ← read\_record(group.data\_array[pos])

if val = empty

val ← get\_from\_buffer(group.buf, key)

if val = empty && group.tmp\_buf , null

val ← get\_from\_buffer(group.tmp\_buf, key)

return val

（2）put(root, key, val):

retry:

group ← get\_group(root, key)

pos ← get\_position(group, key)

if pos !=empty

if update\_record(group.data\_array[pos], val) = true

return

if group.buf\_frozen = false

insert\_buffer(group.buf, key, val)

else

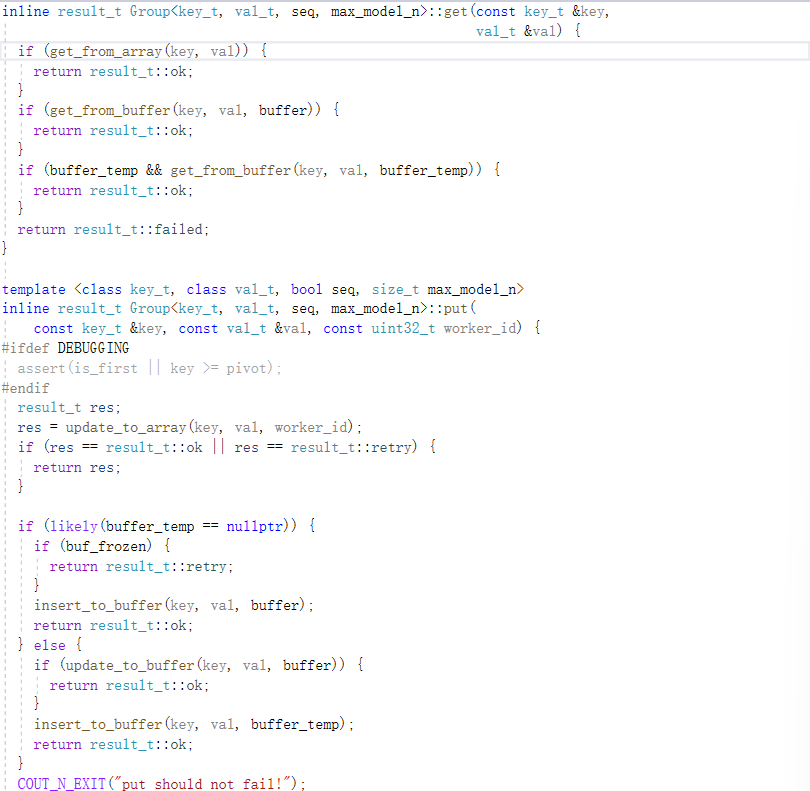
if update\_in\_buffer(group.buf, key, val) = true

return

if group.tmp\_buf = null

goto retry

insert\_buffer(group.tmp\_buf, key, val)



4.3 Two-Phase Compaction：

compact(group):

/\* phase 1 \*/

group.buf\_frozen ← true

rcu\_barrier()

group.tmp\_buf ← allocate new delta index

new\_group ← allocate new group

new\_group.data\_array ← merge(group.data\_array,

group.buf )

new\_group.buf ← group.tmp\_buf

train new\_group’s models with new\_group.data\_array

init new\_group’s other fields

old\_group ← group

atomic\_update\_reference(group, new\_group)

rcu\_barrier()

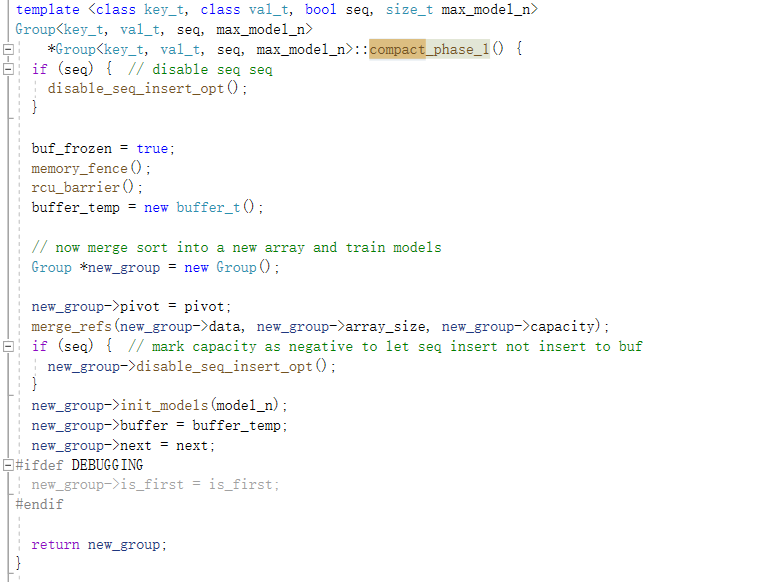
/\* phase 2 \*/

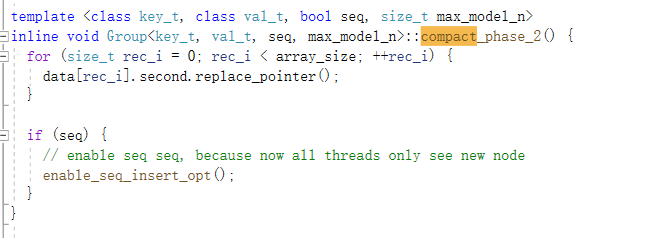
for each record in new\_group.data\_array

replace\_pointer(record)

rcu\_barrier()

reclaim old\_group’s memory





4.4 Group Split：

split(group):

/\* step 1 \*/

g′ a , g′ b ← allocate 2 new group

{ g′ a , g′ b }.{data\_array, buf} ← group.{data\_array, buf}

g′ a .pivot ← group.pivot

g′ b .pivot ← group.data\_array[group.array\_size / 2]

g′ a.next ←g′ b

init other fields of g′ a and g′ b

old\_group ← group

atomic\_update\_reference(group, g′ a)

{ g′ a, g′ b }.buf\_frozen ← true

rcu\_barrier()

{ g′ a, g′ b }.tmp\_buf ← allocate new delta indexes

/\* step 2.1, merge phase \*/

ga, gb ← allocate 2 new groups

tmp\_array ← merge(old\_group.data\_array, old\_group.buf )

{ ga, gb }.data\_array ← split(tmp\_array, g′ b .pivot)

{ ga, gb }.buf ← { g′ a, g′ b }.tmp\_buf

train { ga, gb }’s models with { ga, gb }.data rray

{ ga, gb }.pivot ← { g′ a, g′ b }.pivot

ga.next ←gb

init ga’s and gb ’s other fields

atomic\_update\_reference(group, ga)

rcu\_barrier()

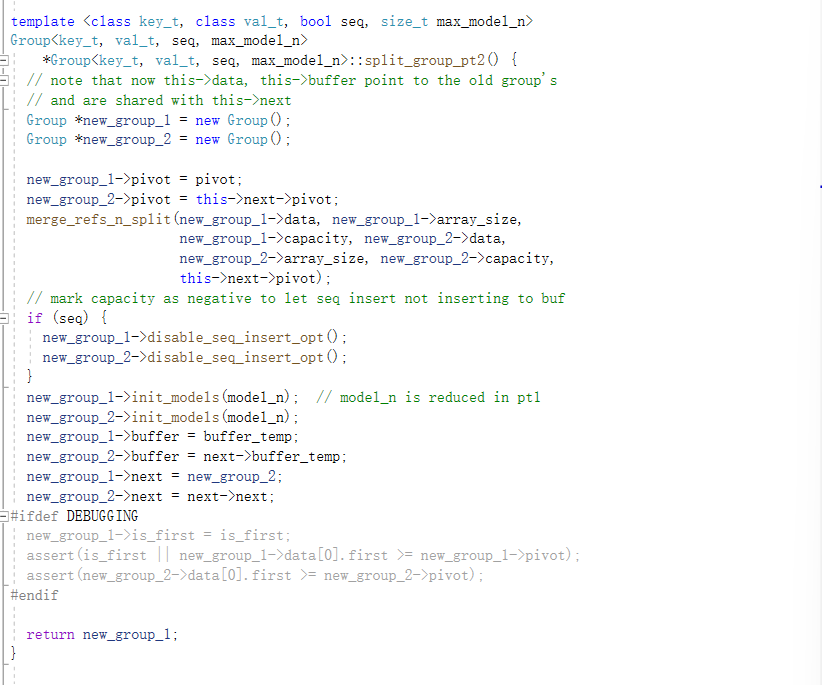
/\* step 2.2, copy phase \*/

for each record in { ga, gb }.data\_array

replace\_pointer(record)

rcu\_barrier()

reclaim {old\_group, g′ a , g′ b }’s memory



4.5 Helper functions：

（1）read\_record(rec):

while true

ver ← rec.ver

removed, is\_ptr, val ← rec.{removed, is\_ptr, val}

if !rec.lock && rec.version = ver

if removed

val ← empty

else if is\_ptr

val ← read\_record(DEREF(val))

return val

（2）update\_record(rec, val):

lock(rec.lock)

succ ← false

if rec.is\_ptr

succ ← update\_record(DEREF(rec.val), val)

else if !rec.removed

rec.val ← val

succ ← true

rec.version ++

unlock(rec.lock)

return succ

（3）replace\_pointer(rec):

lock(rec.lock)

ref.val ← read\_record(DEREF(rec.val))

if ref.val = empty

rec.removed ← true

rec.is\_ptr ← false

rec.version ++

unlock(rec.lock)





第五章 实验结果展示

5.1第一篇论文中的多层神经网及自己设计的其他不同学习索引方法结果展示与对比：

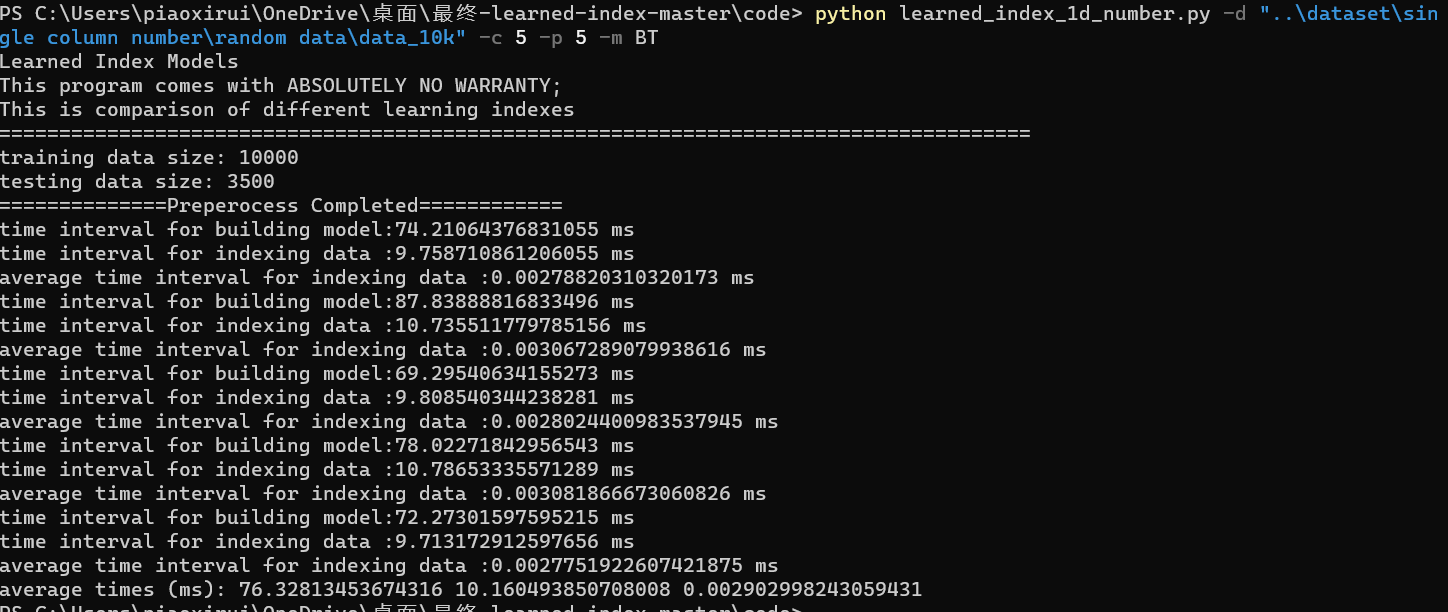
5.1.1 对一维数字在各个模型下的索引：

以下比较的是对同一个数据集"..\dataset\single column number\random data\data\_10k"的各模型结果展示对比，其他数据集也同理运行。

（1）B-Tree模型训练结果：

在code文件夹下运行命令行命令：python learned\_index\_1d\_number.py -d "..\dataset\single column number\random data\data\_10k" -c 5 -p 15 -m BT

即在单列大小10k数据集random data下采用运行五次代码，输入的超参数B树节点大小为5，调用BT模型训练和对测试集检验的效果如下：

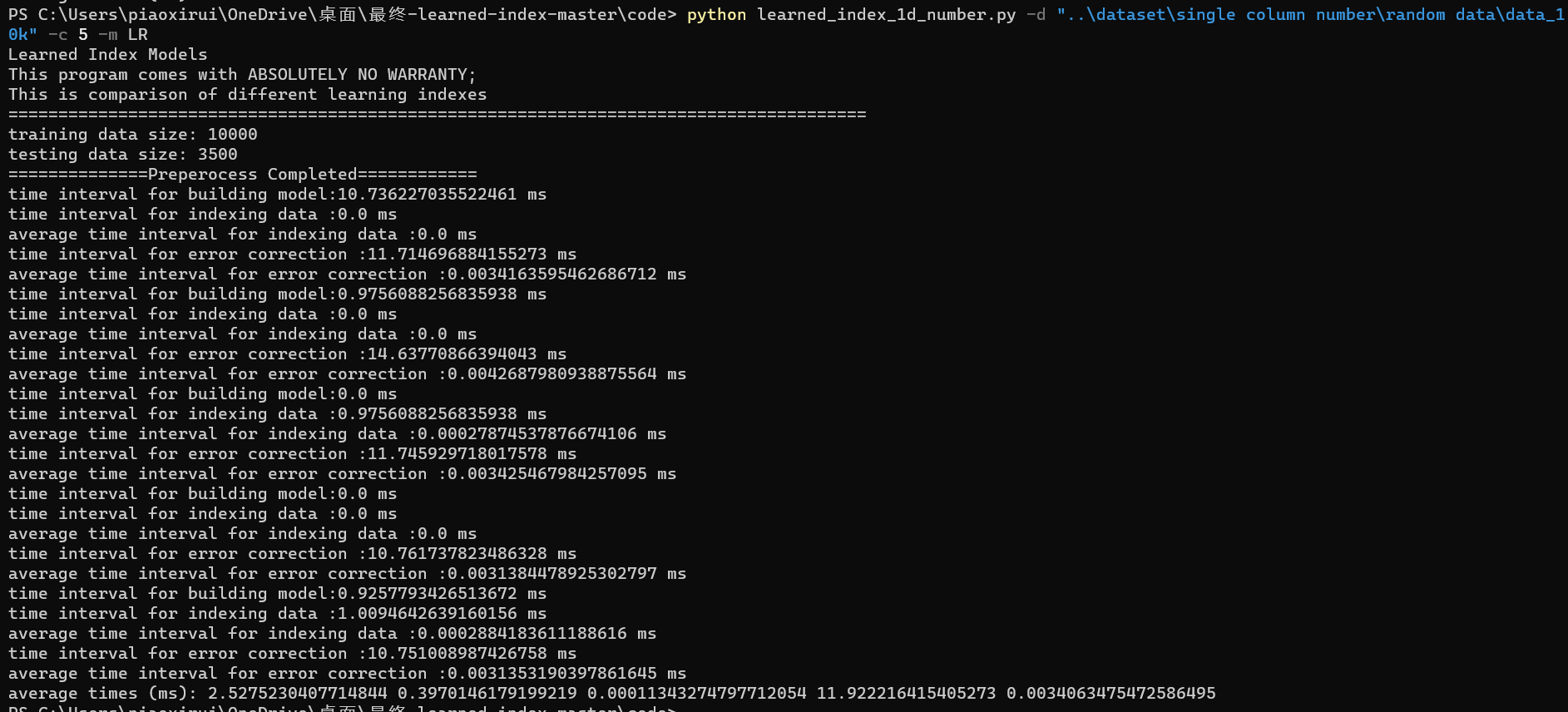


可知在这种情况下此训练模型时间为76.32813453674316ms，对数据记录的索引时间为10.168493850788ms

（2）Linear Regression模型训练结果：

在code文件夹下运行命令行命令：python learned\_index\_1d\_number.py -d "..\dataset\single column number\random data\data\_10k" -c 5 -m LR

即在单列大小10k数据集random data下采用运行五次代码，调用LR模型训练和对测试集检验的效果如下：

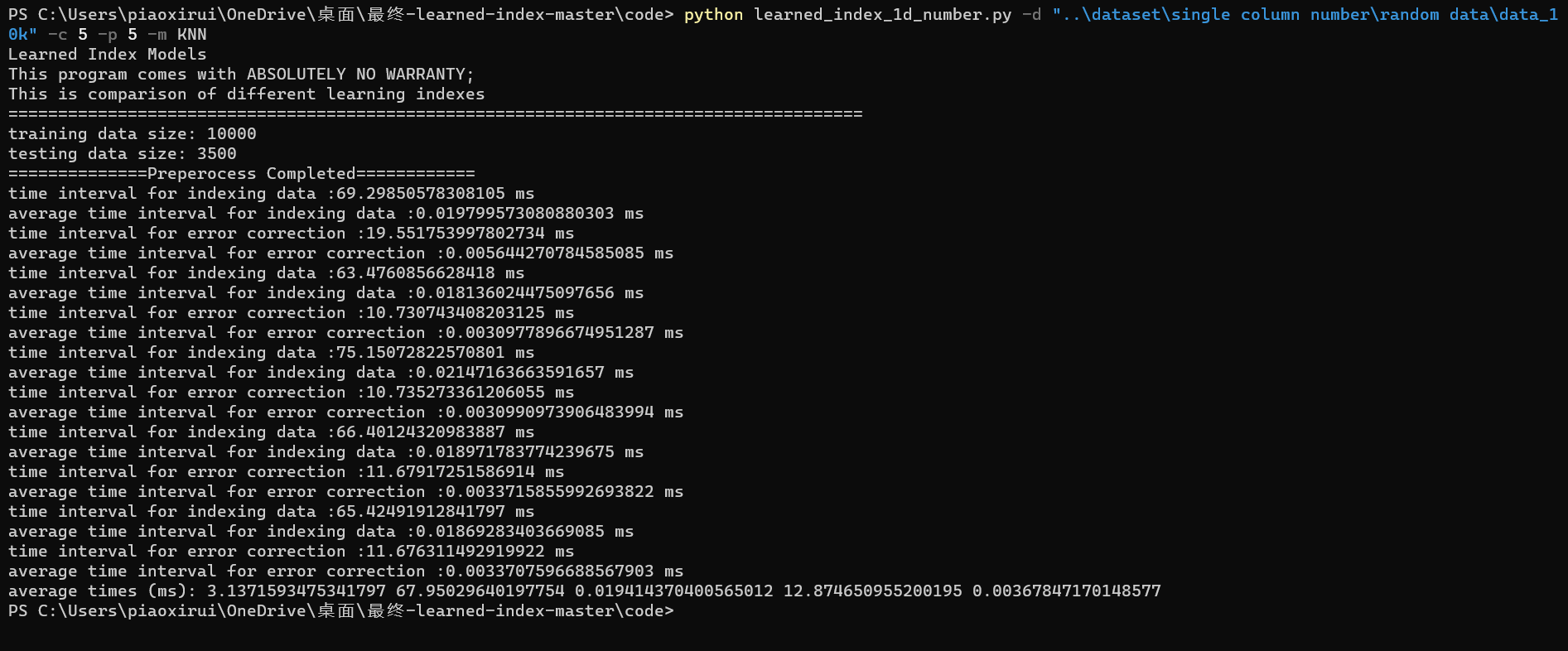


可知在这种情况下此训练模型时间为2.5275230407714844ms，对数据记录的索引时间为0.397014611791ms

（3）KNN模型训练结果：

在code文件夹下运行命令行命令：python learned\_index\_1d\_number.py -d "..\dataset\single column number\random data\data\_10k" -c 5 -p 5 -m KNN

即在单列大小10k数据集random data下采用运行五次代码，输入的超参数k为5，调用KNN模型训练和对测试集检验的效果如下：

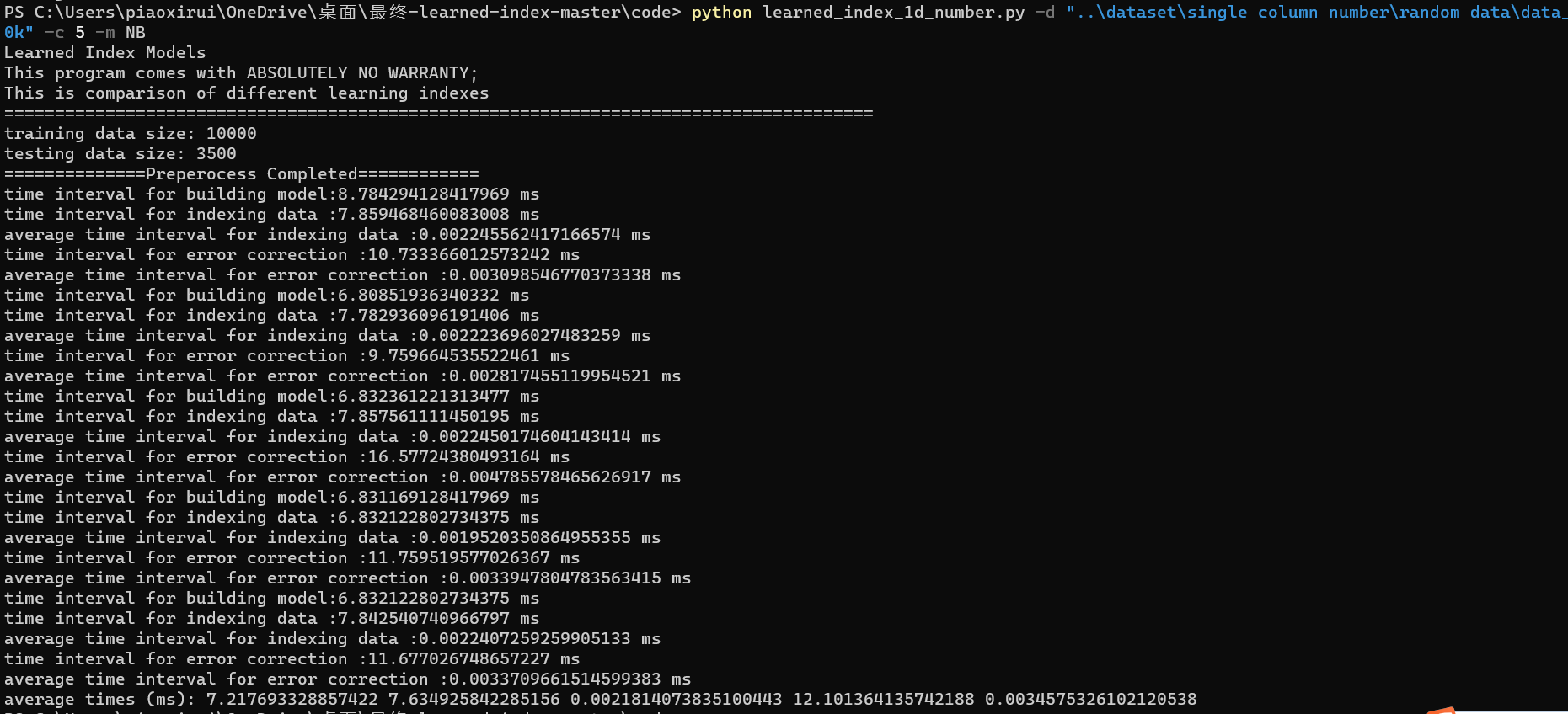


可知在这种情况下此训练模型时间为3.1371593475ms，对数据记录的索引时间为67.95029640197754ms

（4）Naïve Bayes模型训练结果：

在code文件夹下运行命令行命令：python learned\_index\_1d\_number.py -d "..\dataset\single column number\random data\data\_10k" -c 5 -m NB

即在单列大小10k数据集random data下采用运行五次代码，调用NB模型训练和对测试集检验的效果如下：

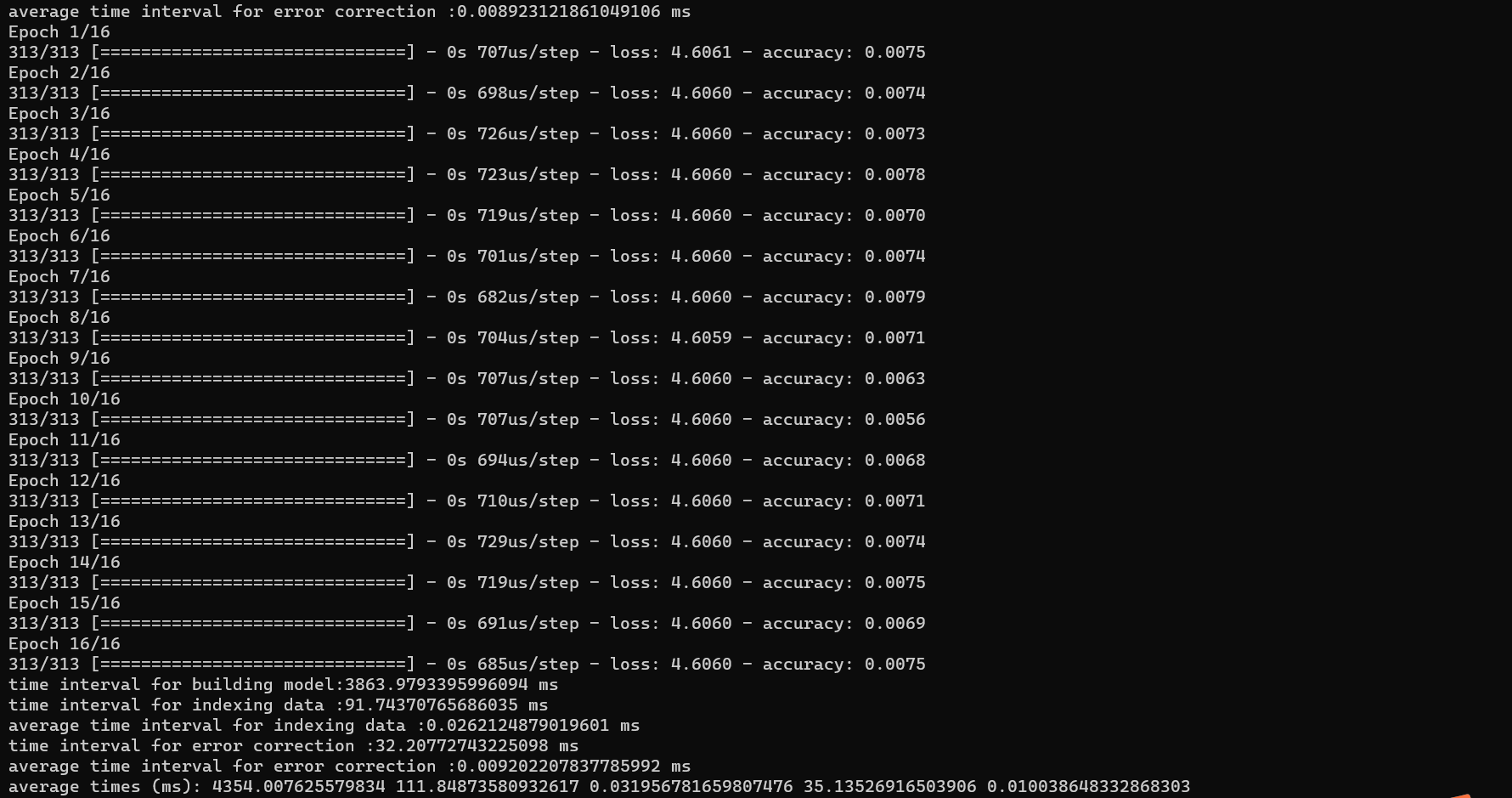


可知在这种情况下此训练模型时间为7.217693328857422ms，对数据记录的索引时间为7.634925842285ms

（5）Neural Networks模型训练结果：

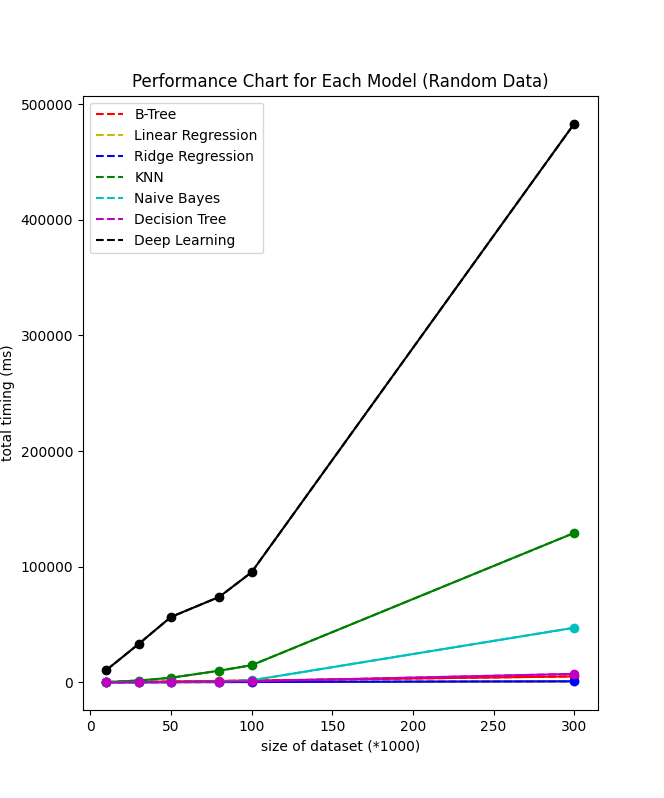
在code文件夹下运行命令行命令：python learned\_index\_1d\_number.py -d "..\dataset\single column number\random data\data\_10k" -c 5 -m NN

即在单列大小10k数据集random data下采用运行五次代码，调用NN模型训练和对测试集检验的效果如下：



可知在这种情况下此训练模型时间为4354.007625579834ms，对数据记录的索引时间为111.8487358093ms

（6）所有模型在此数据集的对比图：



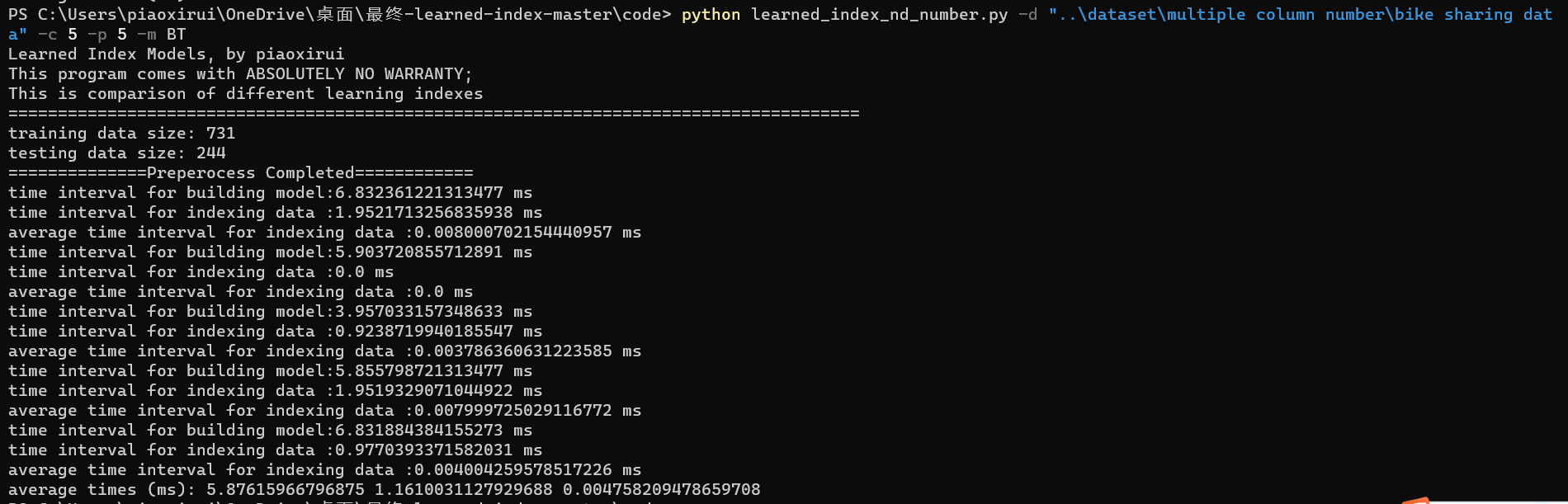
5.1.2对多维数字在各个模型下的索引：

以下比较的是对同一个数据集"..\dataset\multiple column number\bike sharing data"的各模型结果展示对比，其他数据集也同理运行。

（1）B-Tree模型训练结果：

在code文件夹下运行命令行命令：python learned\_index\_nd\_number.py -d "..\dataset\multiple column number\bike sharing data" -c 5 -p 5 -m BT

即运行五次代码，输入的超参数B树节点大小为5，调用BT模型训练和对测试集检验的效果如下：

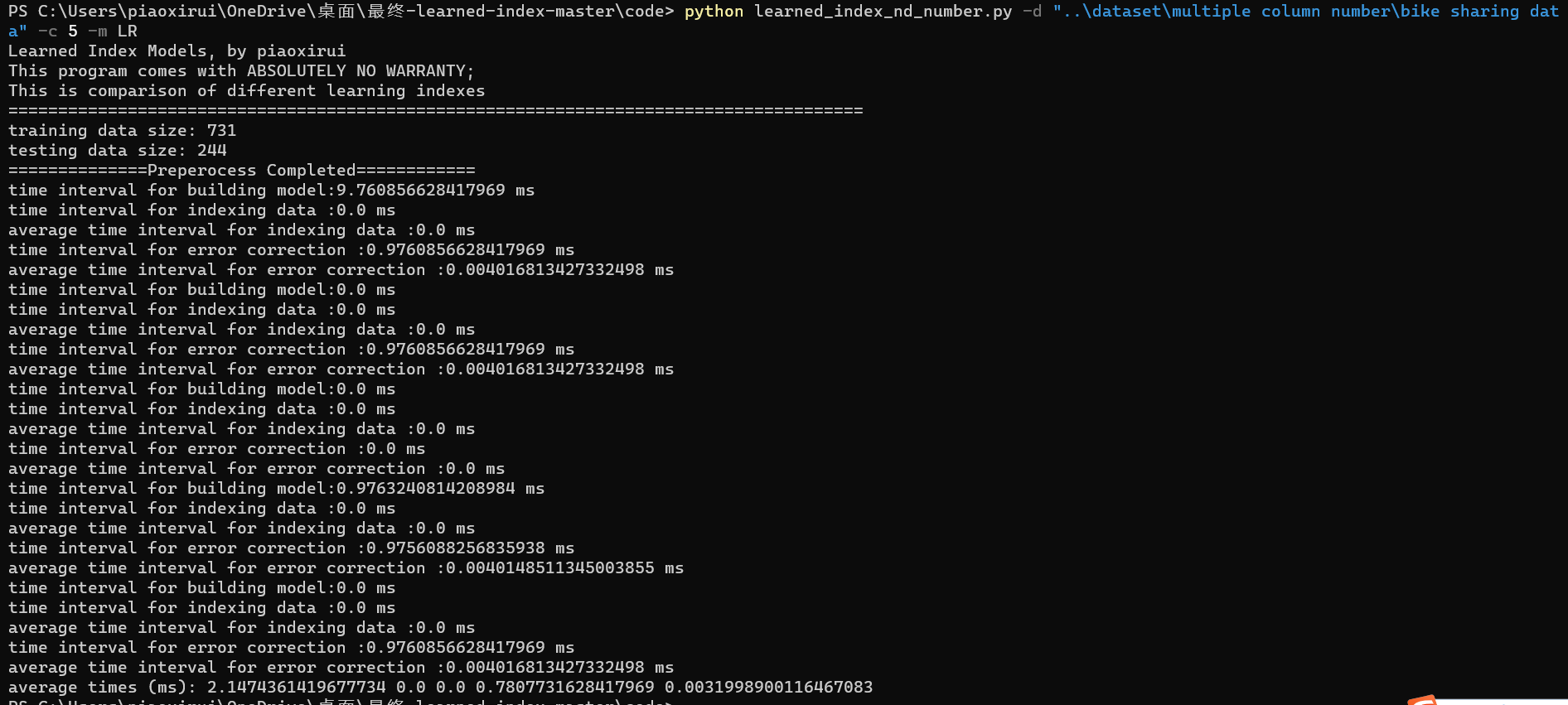


可知在这种情况下此训练模型时间为5.87615966796875ms，对数据记录的索引时间为1.16100311279ms

（2）Linear Regression模型训练结果：

在code文件夹下运行命令行命令：python learned\_index\_nd\_number.py -d "..\dataset\multiple column number\bike sharing data" -c 5 -m LR

即运行五次代码，调用LR模型训练和对测试集检验的效果如下：

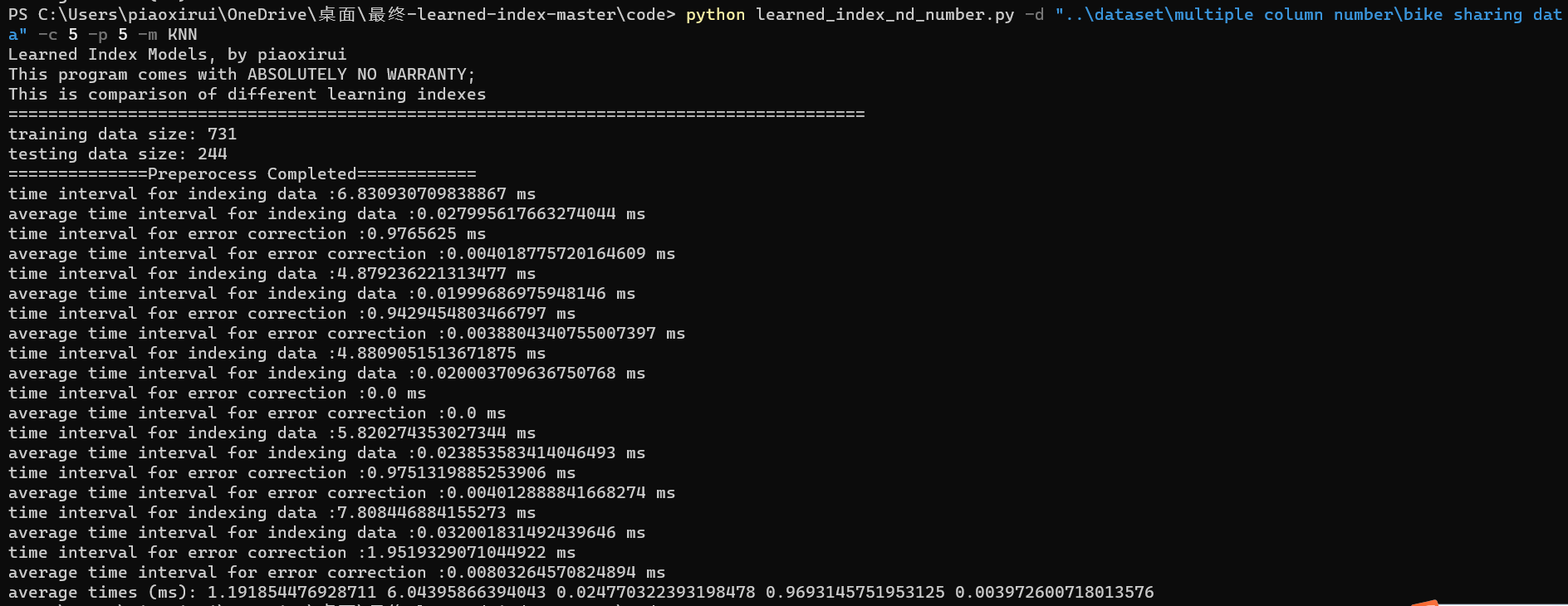


可知在这种情况下此训练模型时间为2.1474361419677734ms，对数据记录的索引时间为将近为0ms

（3）KNN模型训练结果：

在code文件夹下运行命令行命令：python learned\_index\_nd\_number.py -d "..\dataset\multiple column number\bike sharing data" -c 5 -p 5 -m KNN

即运行五次代码，输入的超参数k为5，调用KNN模型训练和对测试集检验的效果如下：

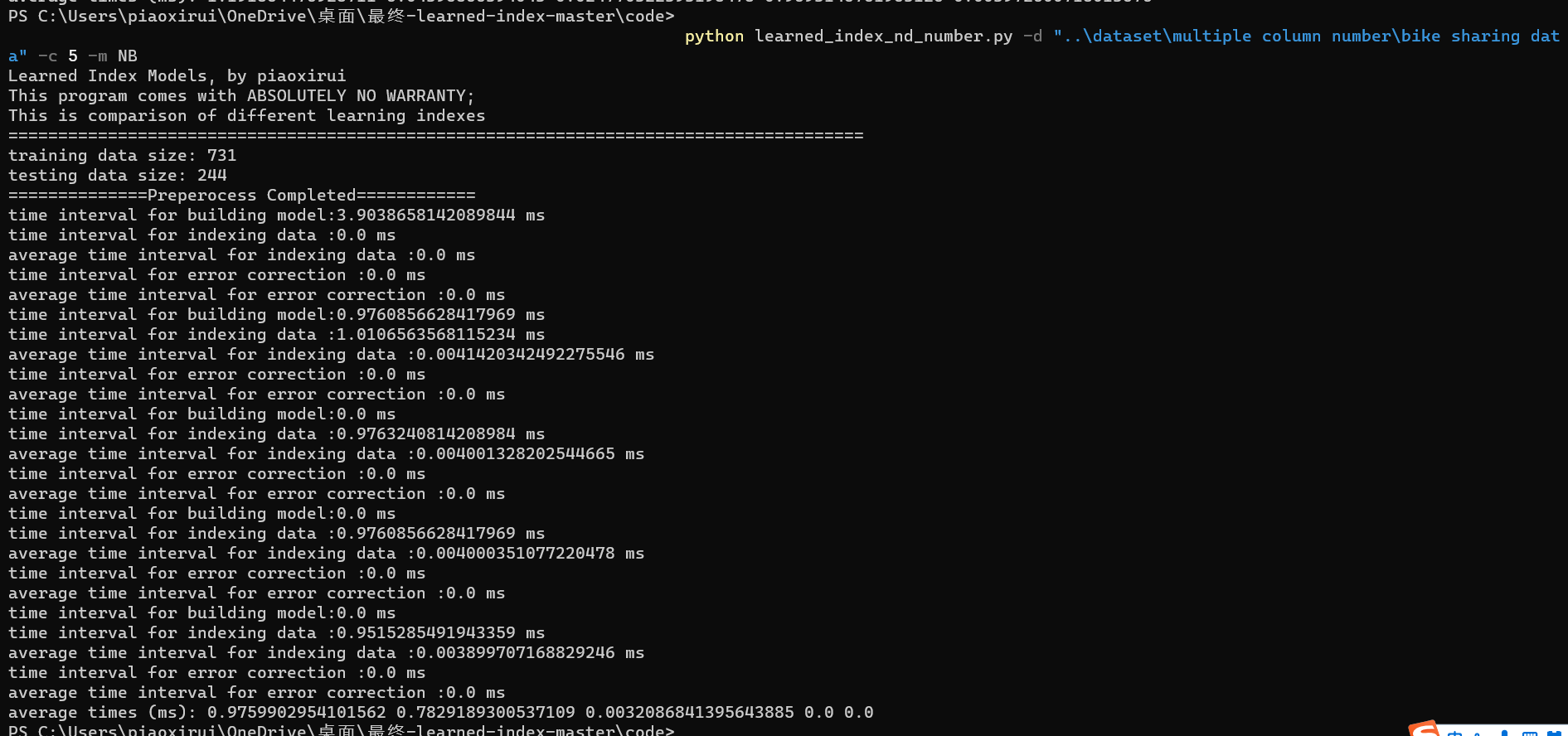


可知在这种情况下此训练模型时间为1.191854476928711ms，对数据记录的索引时间为6.8439586639ms

（4）Naïve Bayes模型训练结果：

在code文件夹下运行命令行命令：python learned\_index\_nd\_number.py -d "..\dataset\multiple column number\bike sharing data" -c 5 -m NB

即运行五次代码，调用NB模型训练和对测试集检验的效果如下：

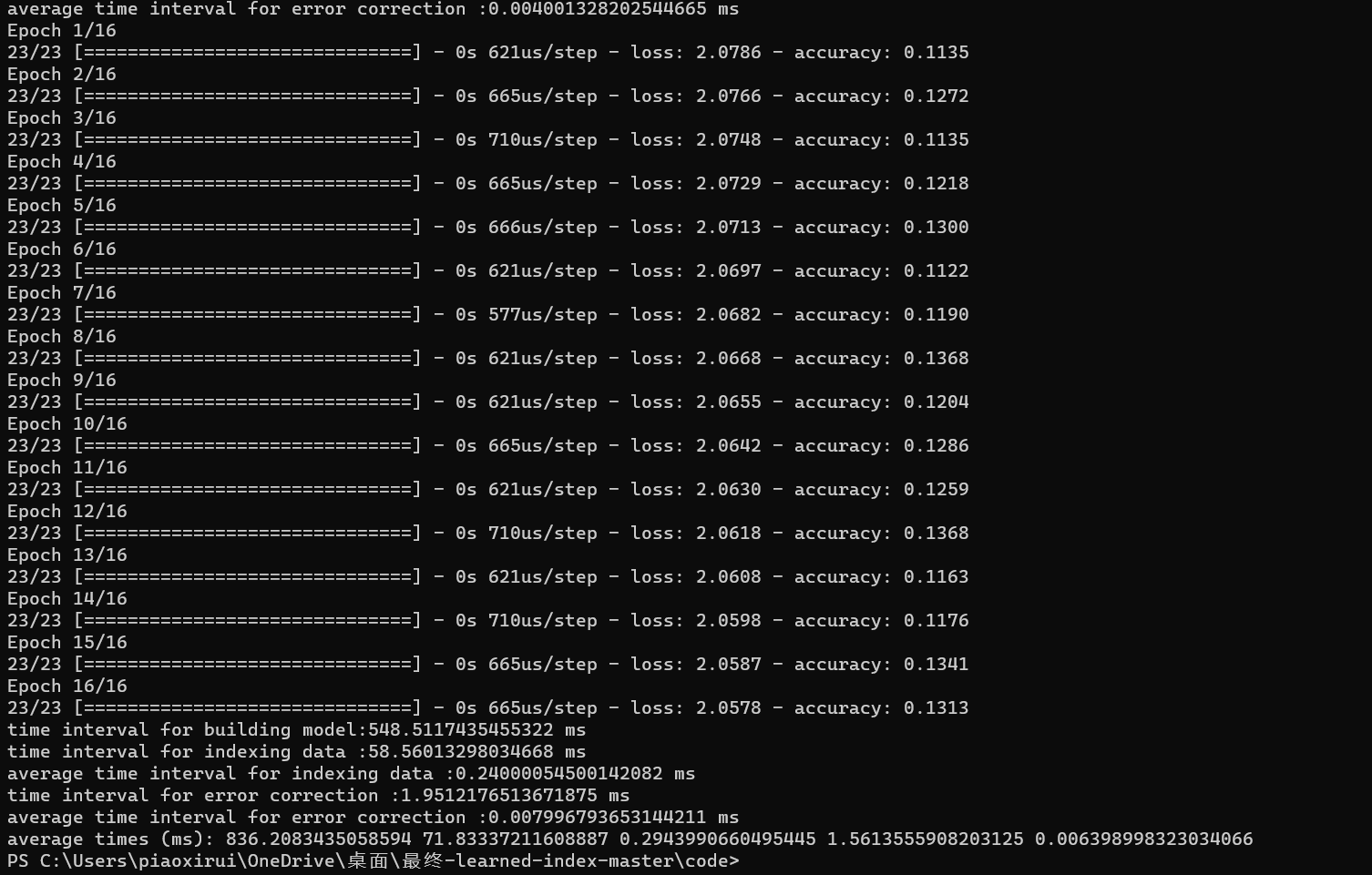


可知在这种情况下此训练模型时间为0.9759902954101562 ms，对数据记录的索引时间为0.7829189300537109 ms

（5）Neural Networks模型训练结果：

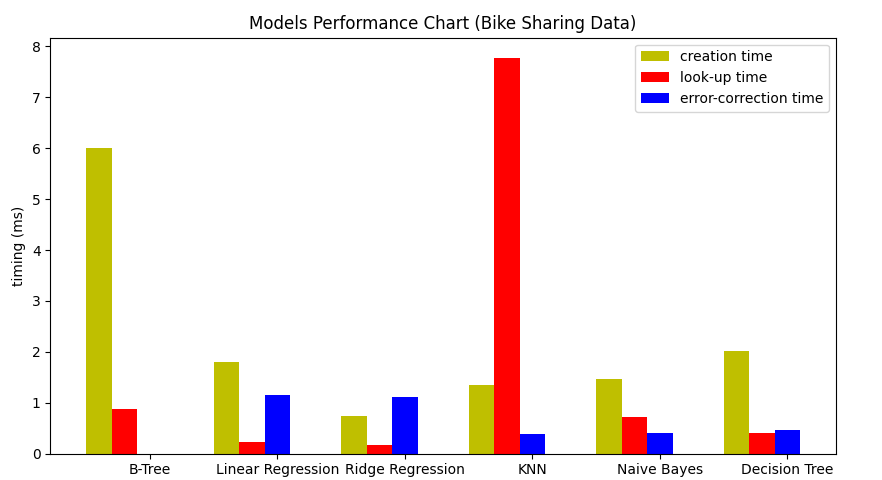
在code文件夹下运行命令行命令：python learned\_index\_nd\_number.py -d "..\dataset\multiple column number\bike sharing data" -c 5 -m NN

即运行五次代码，调用NN模型训练和对测试集检验的效果如下：



可知在这种情况下此训练模型时间为836.2083435058594 ms，对数据记录的索引时间为71.83337211608887 ms

（6）所有模型在此数据集的对比图：



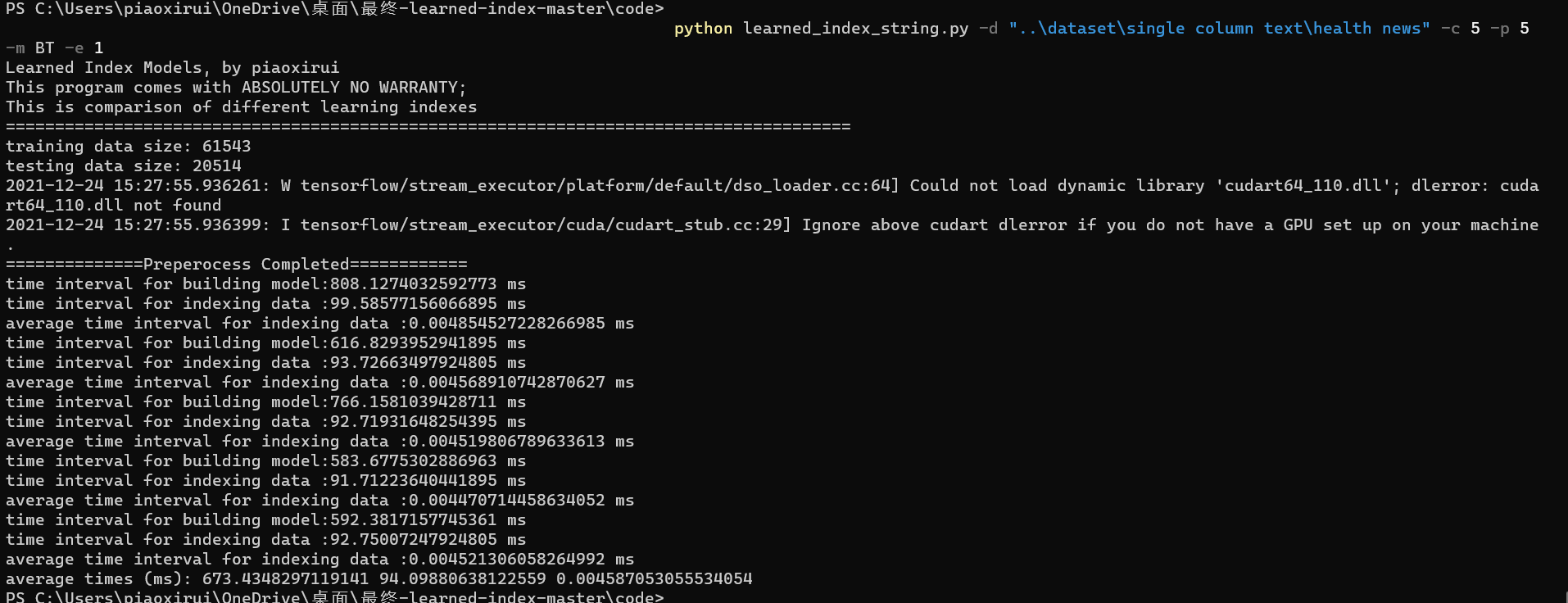
5.1.3对一维字符串在各个模型下的索引：

以下比较的是对同一个数据集"..\dataset\single column text\health news"的各模型结果展示对比，其他数据集也同理运行。

（1）B-Tree模型训练结果：

在code文件夹下运行命令行命令：python learned\_index\_string.py -d "..\dataset\single column text\health news" -c 3 -p 5 -m BT -e 1

即运行3次代码，输入的超参数B树节点大小为5，调用BT模型训练和对测试集检验的效果如下：

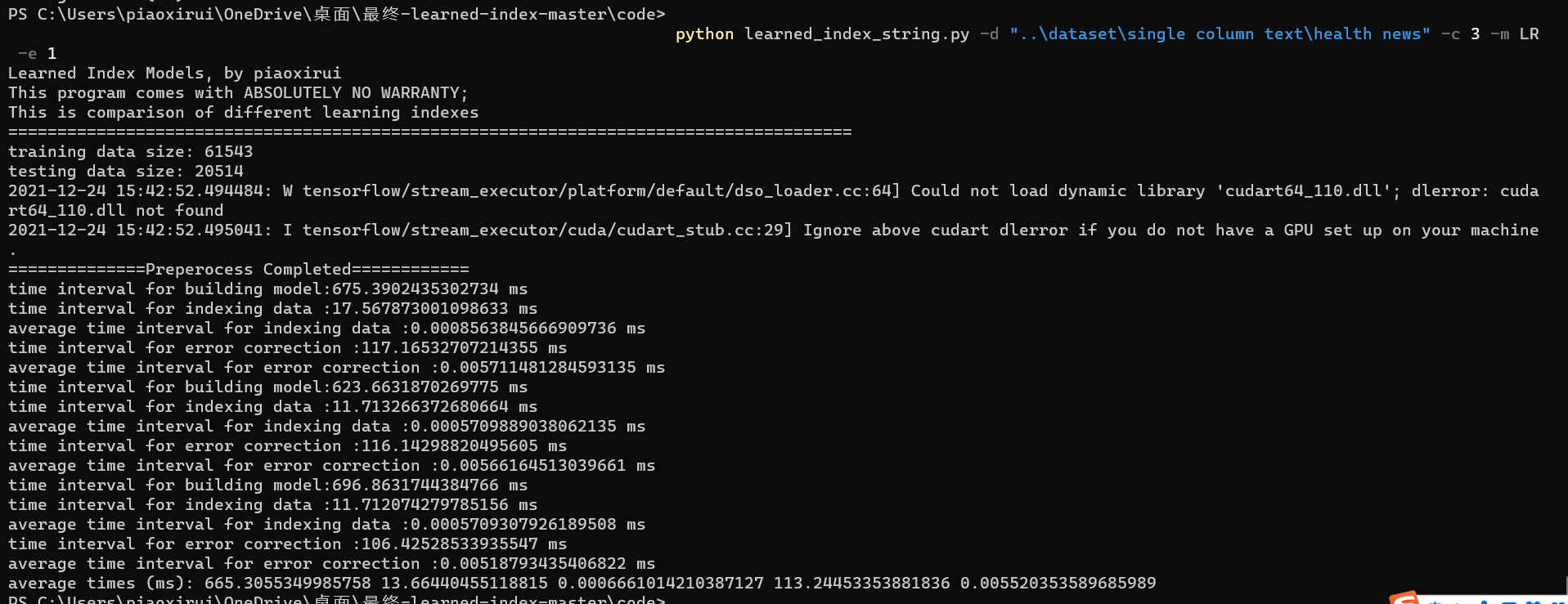


可知在这种情况下此训练模型时间为673.4348297119141 ms，对数据记录的索引时间为94.09880638122559 ms

（2）Linear Regression模型训练结果：

在code文件夹下运行命令行命令：python learned\_index\_string.py -d "..\dataset\single column text\health news" -c 3 -m LR -e 1

即运行3次代码，调用LR模型训练和对测试集检验的效果如下：

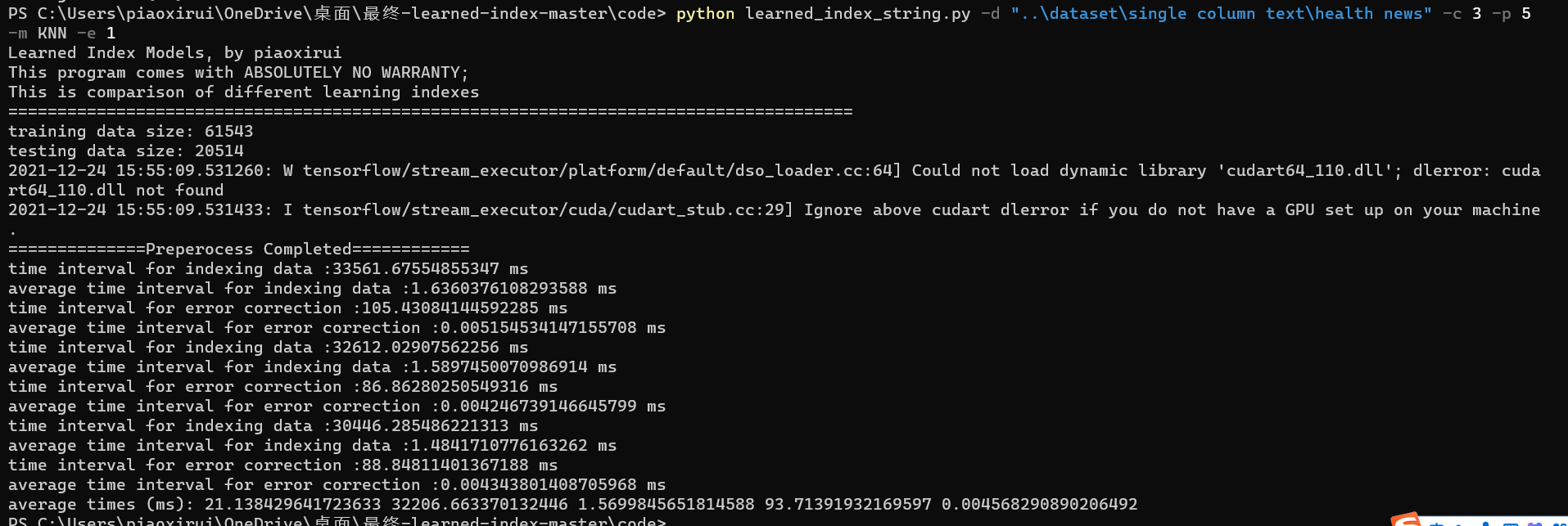


可知在这种情况下此训练模型时间为665.3055349985758 ms，对数据记录的索引时间为将近为13.66440455118815 ms

（3）KNN模型训练结果：

在code文件夹下运行命令行命令：python learned\_index\_string.py -d "..\dataset\single column text\health news" -c 3 -p 5 -m KNN -e 1

即运行3次代码，输入的超参数k为5，调用KNN模型训练和对测试集检验的效果如下：

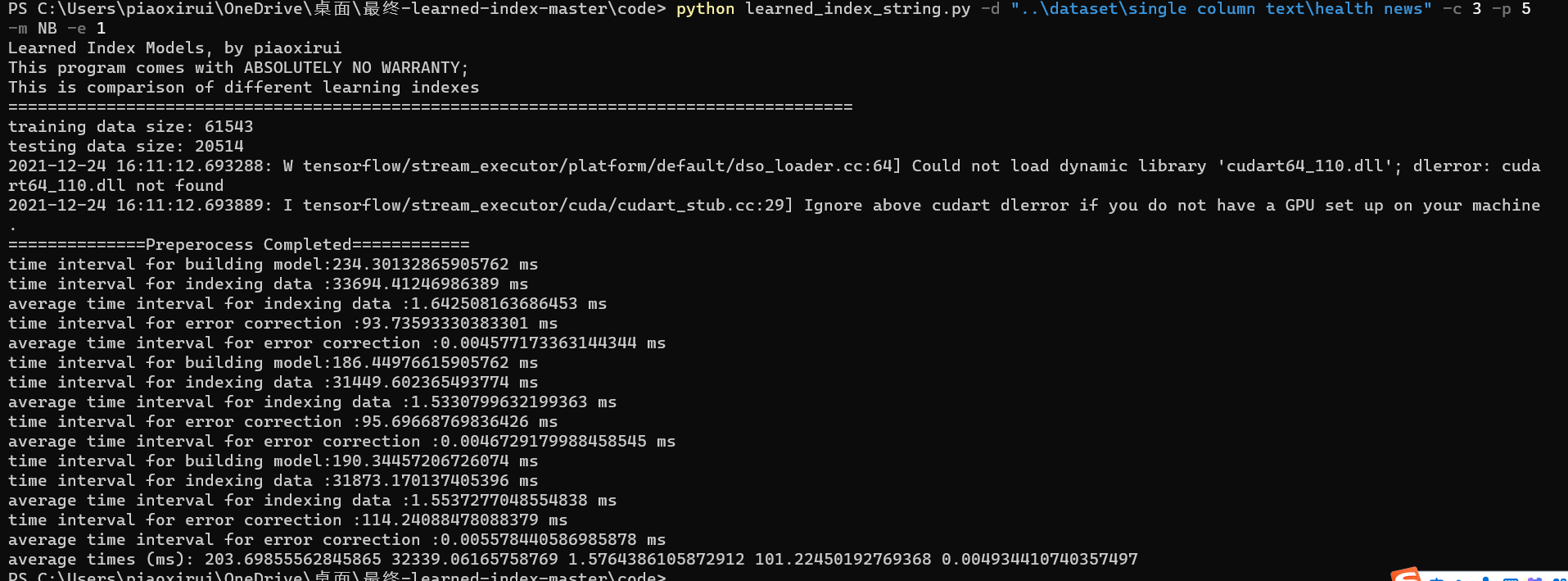


可知在这种情况下此训练模型时间为21.138429641723633 ms，对数据记录的索引时间为32206.663370132446 ms

（4）Naïve Bayes模型训练结果：

在code文件夹下运行命令行命令：python learned\_index\_string.py -d "..\dataset\single column text\health news" -c 3 -m NB -e 1

即运行3次代码，调用NB模型训练和对测试集检验的效果如下：

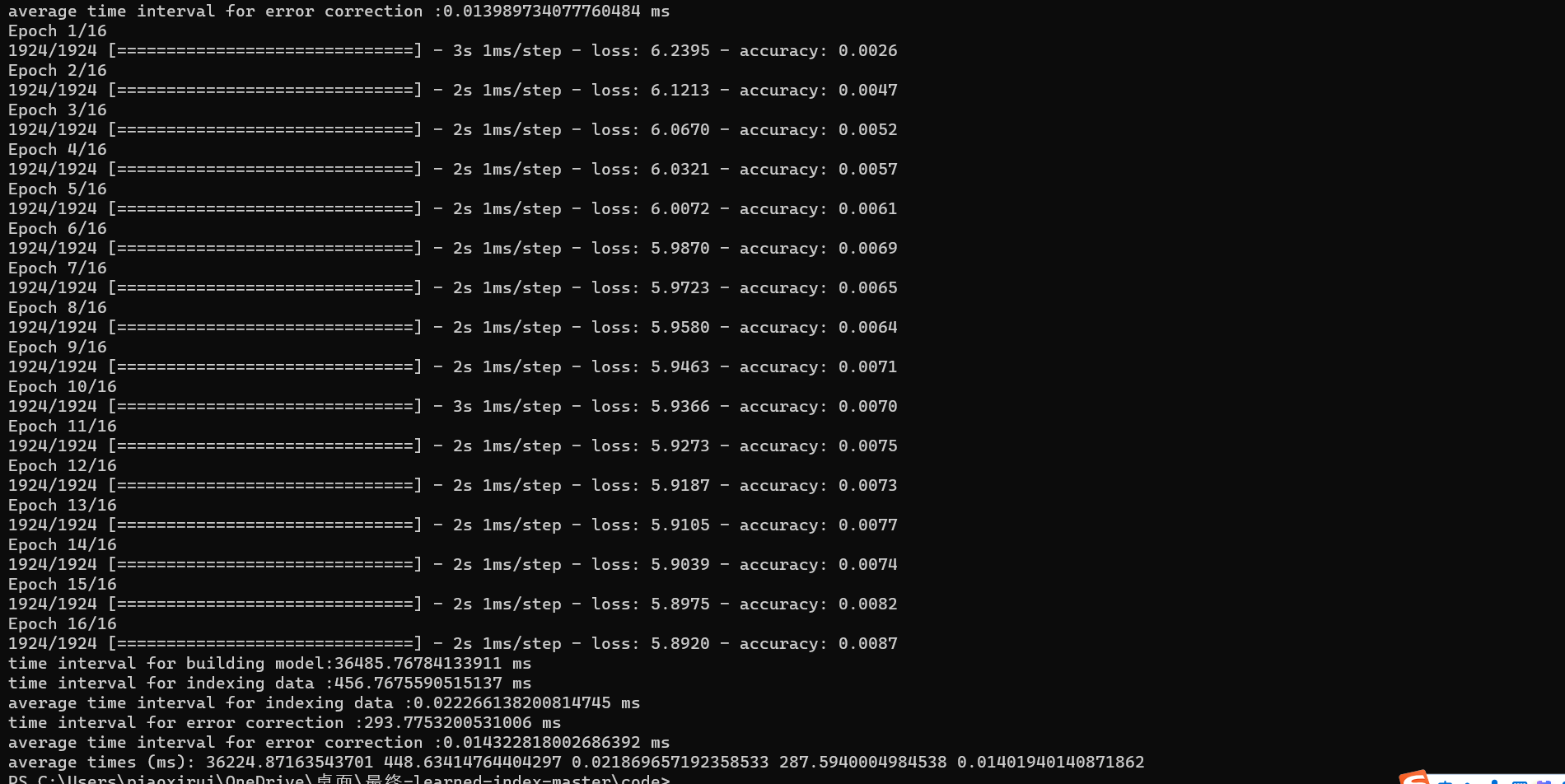


可知在这种情况下此训练模型时间为203.69855562845865 ms，对数据记录的索引时间为32339.06165758769 ms

（5）Neural Networks模型训练结果：

在code文件夹下运行命令行命令：python learned\_index\_string.py -d "..\dataset\single column text\health news" -c 3 -m NN -e 1

即运行3次代码，调用NN模型训练和对测试集检验的效果如下：



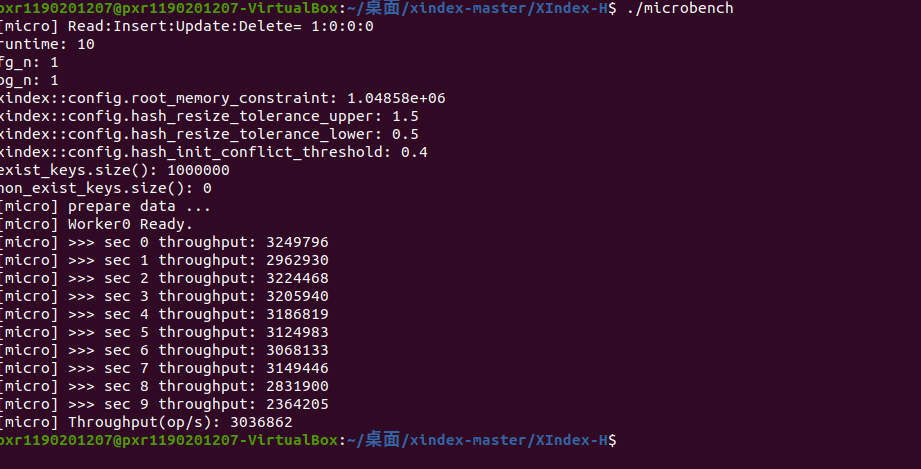
可知在这种情况下此训练模型时间为36224.87163543701 ms，对数据记录的索引时间为448.63414764404297 ms

根据以上及其他数据集上的结果可以发现不管数据集是什么类型的，当数据集较小时，由于学习索引如神经网模型因为训练过程时间长所以加上查询时间后的总时间并不比传统的索引方法如B树好；而随着数据集大小的增加，因为模型训练时间不变，所以学习索引的优势发挥得越来越明显。

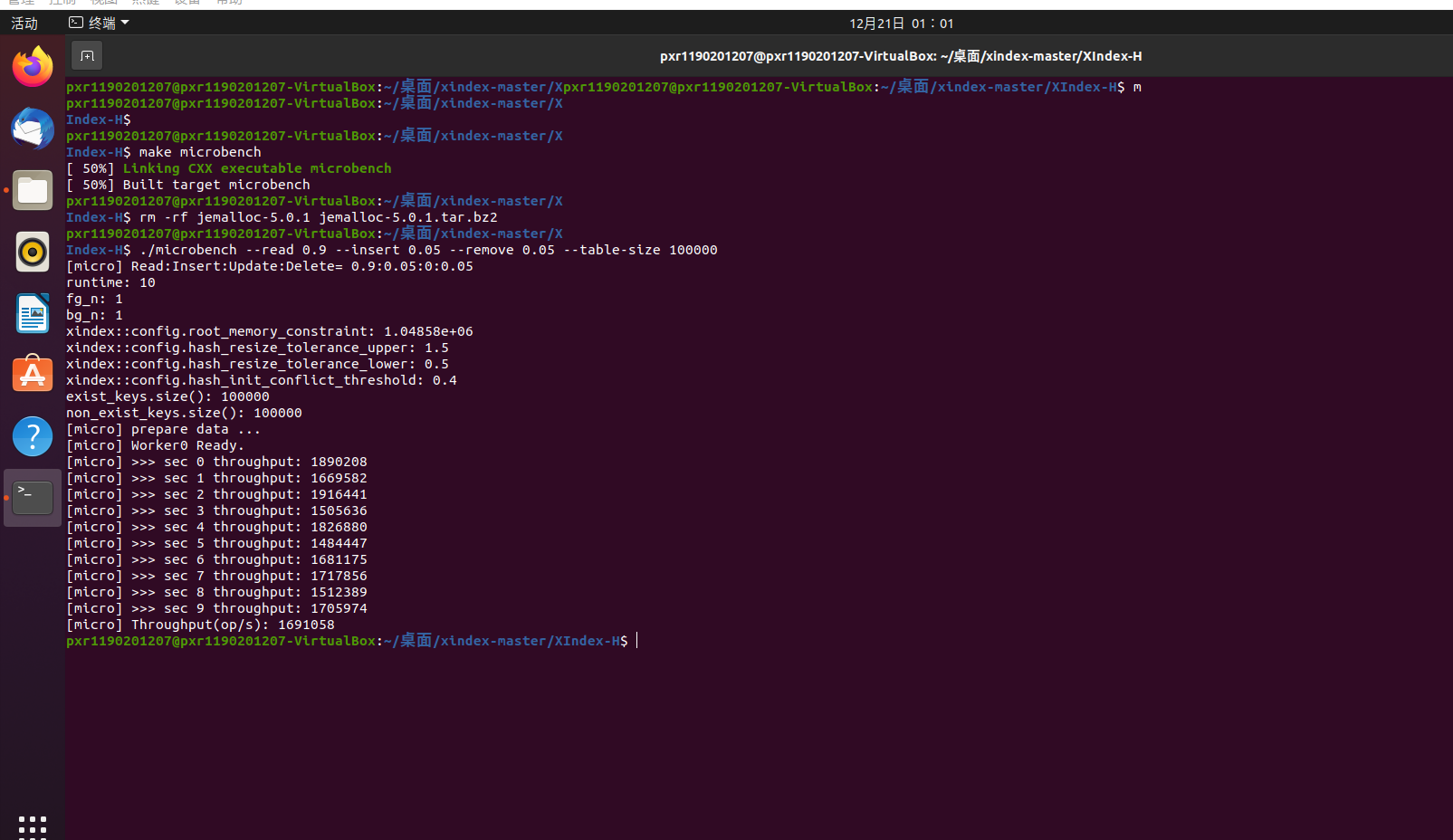
5.2第二篇论文中的并行学习索引方法XIndex的结果展示：

5.2.1：并发学习点索引XIndex-H的结果展示：

当所有操作都为读操作时，进行微基准测试的吞吐率结果如下：



当包括插入，移除等操作时，即支持数据更新的方法，吞吐率结果如下：

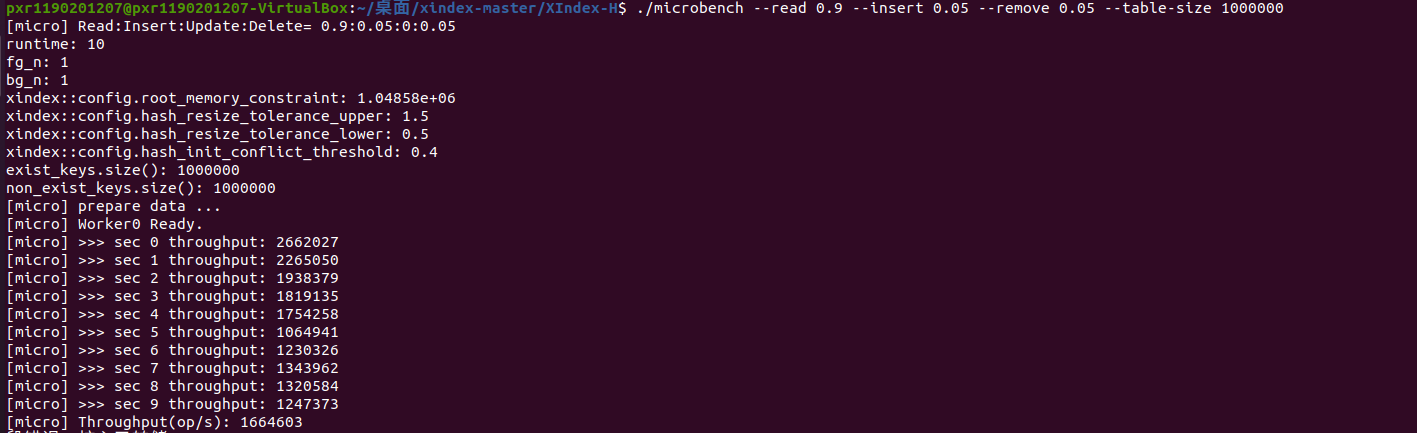


5.2.2：并发学习范围索引XIndex-R的结果展示：

当所有操作都为读操作时，进行微基准测试的吞吐率结果如下：



当包括插入，移除等操作时，即支持数据更新的方法，吞吐率结果如下：



第六章 相关工作

6.1 B树和变体：

在过去的几十年中，已经提出了各种不同的索引结构。然而，所有这些方法都与学习索引的思想是正交的，因为他们都没有从数据的分布中学习，以实现更紧凑的指数表示或性能收益。 与此同时，正如我们的混合索引，可以将现有的硬件觉察索引策略与学习模型更紧密地结合起来，以获得进一步的性能提升。  
 由于B +树消耗大量内存，所以在压缩索引方面也有很多工作，如前缀/后缀截断，字典压缩，键(Key)规范化。 有趣的是，一些现有的压缩技术与我们的方法相辅相成，可以帮助进一步提高效率。 例如，字典压缩可以看作是一种嵌入形式（即将一个字符串表示为一个唯一的整数）。然而，学习索引更进一步，并建议使用学习模型替换整个索引结构。

6.2更好的Hash函数：

类似于基于树的索引结构的工作，在散列图和散列函数方面有很多工作明确映射到哈希映射内的有限插槽集合的目标。

6.3 Bloom-Filters：

最后，我们的存在索引直接建立在Bloom-Filters的现有工作上。 然后通过提出一个bloom-filter增强分类模型或者使用模型作为特殊的散列函数，来对问题采取不同的观察方式，这里的优化目标与我们为Hash Map创建的散列模型完全不同。

6.4简洁数据结构：

在学习索引和简洁数据结构之间存在着一个有趣的联系，特别是诸如小波树的排序选择字典。 然而，许多简洁的数据结构集中于H0熵（即，对索引中的每个元素进行编码所需的位数），而学习的索引尝试学习基础数据分布以预测每个元素的位置。 因此，学习的索引可能实现更高的压缩率。对于每个用例，通常都必须仔细构造简洁的数据结构，而学习的索引通过机器学习“自动化”这一过程，简洁数据结构可能为进一步研究学习指标提供了一个框架。

6.5 CDF建模：

我们的范围和点索引模型都与累积分布函数（CDF）的模型密切相关。 CDF计算是个已经在机器学习有了大多数研究，集中在对概率分布函数（PDF）进行建模，留下许多关于如何有效建模CDF的开放式问题。

6.6 学习索引近年来的发展方向：

已经有人在学习到的索引基础上扩展或构建系统，许多著作扩展了学习索引以支持写入。ALE通过在已排序的数组中为插入新数据预留槽来实现这一点，它分配一个新数组，并在没有足够的预留槽时同步地重新训练模型；AIDEL通过为排序数组中的每条记录附加一个排序列表来处理插入，当列表太长时，它将数据复制到一个新的排序数组中，并同步地重新训练模型。这两个数据结构都不是为并发场景设计的，在重新安排数据和重新训练模型时，操作会被阻塞。

PGM-index扩展了学过的index，针对给定的时空权衡优化了结构。它递归地构造索引结构，并提供最优数量的线性模型。与PGM-index相比，XIndex在运行时调整其结构，不假设已知的查询分布。SageDB是一个数据库，它建议利用学习到的索引进行数据索引，并加速排序和连接。拟合树通过混合B-tree和分段线性函数对数据进行索引，使其成为学习到的索引的变体。它支持插入并提供严格的错误保证。与FITing-Tree相比，XIndex是一种成熟的并发索引结构，它的结构可以适应运行时的数据和查询分布。

第七章 结论

通过实现第一篇论文及其他独立设计的学习索引方法，结论是通过利用被索引的数据分布，学习索引相比传统的索引模型可以提供显著的好处。同时由实验结果发现，学习索引在多维数据上索引也表现的很好，可以说学习型索引思想最令人兴奋的研究方向是将其扩展到多维索引。神经网络模型非常擅长捕捉复杂的高维关系。根据以上及其他数据集上的结果可以发现不管数据集是什么类型的，当数据集较小时，由于学习索引如神经网模型因为训练过程时间长所以加上查询时间后的总时间并不比传统的索引方法如B树好；而随着数据集大小的增加，因为模型训练时间不变，所以学习索引的优势发挥得越来越明显。

通过借鉴论文《XIndex: A scalable learned index for multicore data storage》作者的思路实现一种基于学习索引的并发、灵活的索引结构XIndex。XIndex通过结合创新的两阶段压缩和许多经典并发技术，在多核平台上实现了高性能。此外，它可以根据运行时的工作负载动态调整其结构，以保持竞争性能，并且也可以支持数据更新的索引工作，广泛的评估表明，与Masstree和Wormhole相比，XIndex可以有高达3.2×和4.4×的性能优势。

第八章 参考文献

[1] Kraska T, Beutel A, Chi EH, Dean J, Polyzotis N. The case for learned index structures. In: Proc. of the Int. Conf. on Management of Data. 2018. 489-504.

[2] Tang C, Wang Y, Hu G, Dong Z, Wang Z, Wang M. XIndex: A scalable learned index for multicore data storage. In: ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. 2020. 308-320.

[3] 数据库中的机器学习：哈希表、B 树与学习索引

网址：https://yey.world/2020/08/21/LearnedIndex-02/

[4] <https://ipads.se.sjtu.edu.cn:1312/opensource/xindex> Chuzhe Tang对论文《XIndex: A scalable learned index for multicore data storage》的实现